Towards Highly Available and Self-Healing Grid Services

Stefania Costache, Thomas Ropars, Christine Morin

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Towards Highly Available and Self-Healing Grid Services

Stefania Costache — Thomas Ropars — Christine Morin

N° 7264

April 2010

ISSN 0249-6399 ISRN INRIA/RR--7264--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
RENNES - BRETAGNE ATLANTIQUE

Towards Highly Available and Self-Healing Grid Services

Stefania Costache*† , Thomas Ropars*† , Christine Morin*†

Thème : Systèmes et services distribués
Équipe-Projet MYRIADS

Rapport de recherche n° 7264 — April 2010 — 51 pages

Abstract: The volatility of nodes in large scale distributed systems endangers the availability of grid services and makes them difficult to use. In such a context, structured peer-to-peer overlays can be used to provide scalable and fault tolerant communication mechanisms. To ensure the availability of services, active replication can be used on top of the overlays. In this paper, we present Semias, a framework that is based on active replication on top of a structured overlay to provide high availability and self-healing for stateful grid services. The self-healing mechanisms of Semias ensure the high availability of the replicated services while minimizing the number of reconfigurations. We have used Semias to make Vigne grid middleware services highly available. Experiments run on Grid'5000 and PlanetLab show the performance and self-healing properties of the framework.

Key-words: Large scale distributed system, Grid Computing, High Availability, Self Healing

This report is based on the master thesis presented by Stefania Costache at Politehnica University of Bucharest, February 2010

* INRIA Rennes-Bretagne Atlantique, Rennes, France
† Stefania.Costache@inria.fr, Thomas.Ropars@inria.fr, Christine.Morin@inria.fr

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Vers des services hautement disponibles et auto-réparants pour grilles

Résumé : La volatilité des nœuds dans les systèmes distribués de grande taille peut compromettre la disponibilité des services de grille et les rendre inutilisables. Dans ce contexte, les réseaux logiques structurés peuvent être utilisés pour fournir une solution de communication passant à l'échelle et tolérante aux fautes. Dupliquer activement les services au dessus du réseau logique permet d'assurer leur haute disponibilité. Dans cet article, nous présentons Semias, une solution fondée sur de la duplication active au dessus d'un réseau logique structuré pour assurer la haute disponibilité et l'auto-réparation de services de grille ayant un état. Les mécanismes d'auto-réparation de Semias assurent la haute disponibilité des services tout en limitant le nombre de reconfigurations. Nous avons utilisé Semias pour rendre des services de l'intergiciel de grille Vigne hautement disponibles. Les expériences menées sur Grid'5000 et PlanetLab démontrent les performances et les capacités d'auto-réparation de notre solution.

Mots-clés : Système distribué de grande taille, Grilles de calcul, Haute disponibilité, Auto-réparation

1 Introduction

A grid gathers a large number of computing resources from different organizations or individuals, spread over a wide geographical area. It is a dynamic environment where resources can be connected or disconnected by their owners at any time. Due to the large scale of the system, the probability of resources failing is high. For example, the statistics made for Grid'5000 [1] by Iosup et. al [2] showed that the resource availability at grid level is around 69% on average, with a mean time between failures of 12 minutes. Thus, the unavailability of resources could have a negative impact on applications that run for a long time and, overall, on the performance of the system. Also, the dynamicity and the large scale of the grid make it hard to use.

The access to grid resources is provided through a grid middleware. To ease grid usage, grid middlewares provide the user with different services, like execution management, data management and resource management [3]. For services to be useful, they have to be available despite the volatility of grid resources. Vigne [4] is such a grid middleware. To cope with the scalability and dynamicity issues, Vigne is based on a structured peer-to-peer overlay. One of Vigne's core components is the application management service. All the applications are run under the control of this service. There is one instance of the service per application. The service instances, called application managers, are placed in a Distributed Hash Table (DHT). The DHT offers good load balancing properties and the peer-to-peer overlay provides fault tolerant key-based routing mechanisms. However, if any of the application managers fails, then the corresponding application runs without control and the information about it is lost. Thus, it is important for the service not to fail during the life of the application. For this, every application manager should be *highly available* and *self-healing*.

High availability is defined as the capacity of a system to satisfy its requirements despite failures [5]. For a service to be highly available, it should have a degree of redundancy such that it is not interrupted when failures occur. Ensuring the high availability can be done through replication. *Self-healing* is the capacity of a system to maintain its degree of fault tolerance. Self-healing is important because in a large scale distributed system it cannot be assumed that a human intervention will quickly handle the consequences of a failure every time. Ensuring self-healing can be done by keeping a constant replication degree for the service.

To make Vigne's application managers highly available, active replication can be used [4]. With this technique, the service is replaced with a group of service's replicas that are all active and process the client's requests. However, building active replication on top of a structured overlay is challenging. The replicas must be always kept consistent, despite node arrivals or failures in the overlay. Having efficient self-healing mechanisms is also something that must be dealt with. To keep the replication degree of the service constant, reconfigurations have to be done automatically. Any node addition, removal, or failure, may lead to a group reconfiguration. A group reconfiguration includes the creation of a replica and a state transfer to keep the created replica consistent with the others. In a dynamic system, there could be many reconfigurations, and many costly state transfers.

In this paper we present a solution to actively replicate stateful services distributed on top of a peer-to-peer overlay. We propose the design of a framework, called Semias, based on the combination of the two techniques. To reconfigure the groups of replicas, we propose self-healing mechanisms based on a set of rules to ensure the availability of the services and to minimize the number of reconfigurations. Then, to replace the group configuration we present a protocol that allows changing all replicas during one reconfiguration with no message loss. We implemented Semias as part of Vigne and used it to replicate Vigne's application managers. We evaluated Semias and the replicated application manager on Grid'5000 and PlanetLab. We show how Semias takes reconfiguration decisions to ensure service's availability despite node arrivals or failures. We also show how the total number of reconfiguration decisions is reduced when managing several application managers.

This paper is organized as follows. Section 2 presents the context of our work, gives an overview of the project, and details the related work. We describe the design of our prototype and detail the reconfiguration management in Section 3. In Section 4, we present our prototype and how we replicate the application management service of Vigne. Section 5 evaluates the performance of Semias. Finally we conclude and suggest some future directions in Section 6.

2 Overview

In this Section we detail the context and the goals of this work. Then, we give a brief overview of the two techniques on which Semias is based. We show how their combination provides services with high availability and we discuss the open issues related to an implementation of active replication on top of a structured peer-to-peer overlay. Finally, we outline the related projects that try to address the same problems as us.

2.1 Context and Motivation

In this Section we give a brief overview of the general context of our work. Then, for a better understanding of our motivation, we describe Vigne, the grid middleware in which we integrated Semias. We outline the main goals that we set for the design of our solution to make distributed services highly available through replication. Finally, we describe the system model that we considered when designing Semias.

2.1.1 Grid Computing

Our work has been carried out in the context of grid computing. A grid gathers a large number of heterogeneous resources owned by different organizations or individuals and it provides collaborative access to them. In general, two types of grids can be distinguished: computational and data grids. Computational grids provide access to great amounts of processing power. Data grids help controlling, sharing and managing large amounts of distributed data. Computational grids can be composed of different distant clusters that belong to different organizations. For example, Grid'5000 [1] or TeraGrid [6] are such grids. Or, they can gather the idle cycles provided by desktops from all the world to provide

large-scale computation and data storage. Folding@Home [7] or Seti@Home [8] are such examples. More generic architectures include both clusters and desktops, like in the case of NorduGrid [9]. Also, they can be composed of clusters and parallel computers, like EGEE [10].

Grid middlewares offer an uniform access to grid resources and a seamless computing environment to the user. XtreemOS [11, 12] and the Globus toolkit [13] are some examples of grid middlewares. A grid middleware provides different capabilities, like security, resource discovery, data storage, execution management, scheduling, etc. These capabilities are provided through grid services, that can be seen as stateful web services [3]. Grid middlewares have to solve several challenges that are related to the grid features. Resources heterogeneity is one of the main challenges. Another challenge is the management of resources spread over wide geographical areas and owned by different organizations. The dynamic nature of the grid poses yet another challenge.

2.1.2 Failures in Grids

Studies have been made to characterize resource availability in grids and other large distributed systems. Parameters like availability time, mean time between failures or mean time to repair are often used to give a clear view regarding resource availability. Mean time to repair, or the failure duration, is the time between the occurrence of a failure and the recovery of the failed resource. Similar, the availability time is the time between the recovery of the resource and its failure. The mean time between failures is the average time between two inherent failures.

Iosup et. al [2] provide an accurate study on the availability of resources in Grid'5000 [1], using traces of Grid'5000 usage from 2005-2006. They found that the mean time between failures was 12 minutes at grid level. Moreover, the average availability of a node was around 45 hours, with an average failure duration of 14 hours. When considering the presence of correlated failures, they found that these failures do not usually expand beyond a site. Also, the authors specify that the chance for a node to become unavailable increases with the period the node stays in the system, thus preventing applications that run for a long time to complete. This fact outlines the need for providing high availability in large scale systems like grids.

2.1.3 Vigne

Vigne is a grid middleware that abstracts for the user the distributed physical resources. It is composed of a set of high-level services like application management, application monitoring, resource discovery, persistent data management, volatile data management, and low-level services like system membership [14]. The application management service controls application execution. The application monitoring service offers information about the execution of applications. The volatile data management service offers distributed shared memory and the persistent data management service is used for storing and transferring files. The resource discovery service is in charge of finding a set of resources corresponding to the job specification. It uses an unstructured peer-to-peer overlay for distributed attribute searching. System membership service connects the nodes

of Vigne through a structured peer-to-peer overlay and is the base for the other services.

Application Management Service Each application is run under the supervision of the application management service. The application management service is a distributed service, with one instance per application. An instance of the service is called an *application manager*. The application manager has the task of running the application and ensuring that it terminates correctly despite nodes failing. For this, it interacts with other components like the client, the resource discovery service and the resource on which the application tasks are run. The application managers are distributed over the grid nodes using the structured peer-to-peer overlay. This provides fault tolerant message routing between applications managers and clients or grid applications. The structured overlay is based on Pastry [15] and uses Bamboo's maintenance algorithms [16]. We describe in Section 2.2.2 the basic principles of Pastry and how is used to distribute the services.

To launch an application, a client contacts one node of Vigne and sends it the job specification. The specification is described in a JSDL [17] (Job Submission Description Language) file and includes the architecture requirements, number of nodes, memory, scheduler, etc. When the application is launched, a random key is associated to it, and an application manager is created for this application on the node whose id is the closest to the key at that moment. Afterward, a client can control the application by sending commands to the application manager, through the peer-to-peer overlay. In order to run an application, an application manager has to execute the following steps. First, it needs to select the nodes on which the application will run. For doing so, the application manager asks the resource discovery service to obtain the necessary resources. Then, if any input files are needed, the application manager asks the file transfer service to transfer them to the nodes on which the application will run. Finally, the tasks are started on the selected nodes and are monitored during their running to ensure their successful execution. If the task fails then the application manager restarts it from the beginning, or from a checkpoint if some checkpointing mechanisms are provided with the application. At the end the application manager cleans the task files, gets the output files and frees the allocated resources.

It is likely that the node on which the application manager resides fails. In this case, the client cannot control the application anymore and the user is not be able to get its results. Also, the resources used by the application will not be freed. Even worse, the nodes on which the application is run could also fail and in this case the application will not complete its execution. To avoid these problems, every application manager should be available even in case of failures.

2.1.4 Goals

We have seen why it is important for a service like Vigne's application management, presented in the previous section, to be highly available. To provide high availability for such a service we want to replicate it. Replicating a service is done by keeping multiple consistent copies (replicas) of it. When the service fails, one of its copies replaces it. However, replication can impact the performance of the system. Keeping multiple replicas leads to the utilisation of more resources and the management of the replicas implies an additional overhead

(the replicas need to be consistent, so they need to be updated when the state of the service changes). Considering these disadvantages, we define some general goals that we would like to address with our work:

- Because we don't want to modify too much the client or the service, and to simplify the work that has to be done to replicate the service, the replication should be transparent. The service or the client should not know about the existence of multiple replicas or how they are managed. In case of failures, the client should not retransmit its requests.
- Because we want services to be usable, the response time of the replicated service should be kept low even in the event of some replica failures.
- Because the number of service instances can be large in the case of a distributed service like Vigne's application management service, the solution for replicating the services should be scalable. We should be able to manage a large number of replicas and distribute their load over the nodes from the system.

2.1.5 System Model

We are considering a large scale distributed system, where nodes can fail or be added at any moment. We don't assume any distribution of the node failure and arrival rate in the system. We assume that nodes crash in a fail-stop manner, i.e. no byzantine failures, and don't recover afterward. Channels are FIFO and reliable but there is no bound on message transmission delay: if a process p_i sends a message to process p_j, and none of them crash, then p_j will eventually receive the message. Regarding group membership, we consider the primary partition model because we want all non-failed replicas of a service to be always consistent.

2.2 Project Overview

Our solution to make distributed stateful services highly available is to use active replication on top of a structured peer-to-peer overlay. The idea of actively replicating services in a DHT for high availability is not new. It was suggested in [4]. The architecture of Mena et. al [18] was proposed for the group communication system used for active replication. However, no further details were given on how this could be done and neither how reconfigurations of the replica groups could be managed. The feasibility of such a solution was investigated afterwards in [19], which gave a qualitative comparison of various replication techniques and outlined the possible open issues that are addressed in this paper.

We present next how this combination between active replication and a structured peer-to-peer overlay meets our previously defined goals. We illustrate how active replication can be implemented in a DHT in the design of Semias. To implement active replication, we started with the existing group communication architecture for dynamic groups, proposed by Mena et.al [18] and we extended it with a reconfiguration protocol to make the groups self-healing.

2.2.1 Replication Techniques

Replication is a common technique for making services fault tolerant. Service replication consists of creating and distributing multiple identical copies (replicas) of the service across the system and keeping them consistent. The two main replication techniques are active and passive replication [20]. The basic principles for these techniques are described in Figure 1.

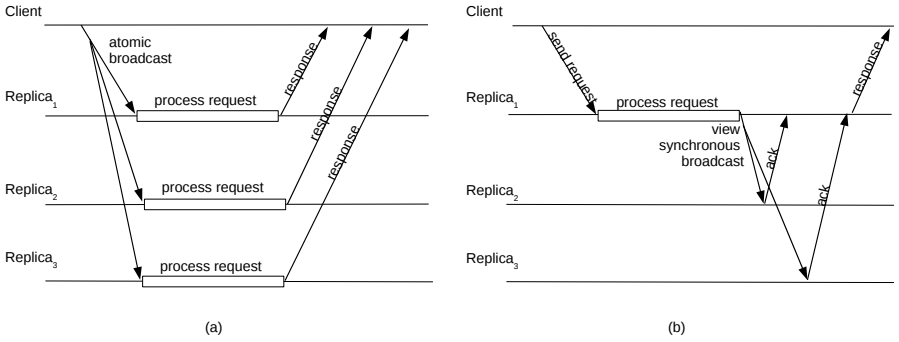


Figure 1: The interactions between the replicas and the client in (a) active replication and (b) passive replication.

In **active replication**, also called state-machine approach [21], every replica processes the client's requests and returns the responses. When processing the requests, the following steps are taken: (i) the requests are sent to all the replicas; (ii) the replicas process them in the order they are delivered; (iii) all replicas send back the responses to the client, which can wait only for the first one, or for a majority. If the client cannot handle duplicate messages, care must be taken to send only one response.

Active replication assumes that if all replicas receive and process the same requests in the same order, they remain consistent. To achieve this consistency, an atomic broadcast group communication primitive [22] is used. Atomic broadcast ensures that the message is delivered at most once and in the same order to all (non-failed) replicas. Active replication requires services to be deterministic. A service is said to be deterministic if started from the same initial state and supplied with the same ordered sequence of messages, reaches the same final state and produces the same output. A strong advantage of the active replication technique is that it ensures the highest degree of availability for the service. In the event of a replica failure, the remaining available replicas continue to process the requests. Thus, the failure is transparent for the clients and the response time of the service remains low.

In **passive replication**, also called primary-backup [23], only one replica, called *the primary*, receives and processes the client requests. After processing a request, the primary sends an update message to the other replicas, called *backups*. All backups update their state and send an acknowledgement message to the primary. After it receives all the acknowledgements, the primary sends the response to the client. If the primary fails, a new primary must be chosen from the available backups. To guarantee that the updates are processed in the same order, a view synchronous broadcast primitive is used. View synchronous

broadcast [24] is defined for dynamic groups. A dynamic group is a set of processes that changes during the system's lifetime. The set of processes at a time t is called a view. When a process is seen as failed or a new process wants to join the group, a new view is installed. View synchronous broadcast ensures that all the processes from a view deliver the same set of messages before installing a new view.

One advantage of passive replication is that it requires less processing power than active replication since the requests are processed only once. Also, it can be used to replicate non-deterministic services. However, passive replication requires modifications of the service to implement the backup update mechanisms. Furthermore, the failure of the primary replica is not transparent for the clients. If the primary fails while processing a client request, the client has to re-send the request. Also, in case of primary's failure there could be a significantly increased response time, due to the time it takes for one of the backups to become the primary and start processing the client's requests again.

Besides active and passive replication, there are also various hybrid techniques, like semi-active [25] or semi-passive [26] replication. **Semi-active replication** tries to accommodate the non-deterministic behavior of the services and to achieve the low recovery overhead of active replication. In semi-active replication, all replicas process the requests but non-deterministic actions are taken only by a leader that notifies the other replicas (followers) of its decisions. **Semi-passive replication** tries to reduce the response time in case of primary's failure. In semi-passive replication all the replicas receive the requests, but only the primary process it and then it updates the backups. Then, the responses are sent by all replicas.

To replicate services, Semias is using **active replication**. We have chosen active replication because it provides the highest degree of availability for the service. Also, the replicated service has a low response time in case of failures and the replication is made transparent to the client. Thus, we found it suitable for our goals. Analyzing Vigne's services, we considered that determinism was not an overly restrictive requirement. However, if we want to handle non-deterministic behaviors [27], we could replace active replication with semi-active replication.

2.2.2 Structured Peer-to-Peer Overlays

Structured peer-to-peer overlays provide key-based routing mechanism. They are used to implement Distributed Hash Tables (DHTs). A DHT is an abstraction that provides a mapping *(key, value)*, in a distributed manner over the nodes of the peer-to-peer network. Several implementations of DHT are described in the literature [28].

Semias is based on Pastry [15], but the design could be also adapted to other DHTs. Pastry provides scalable and fault tolerant routing mechanisms. It uses a ring topology with nodes ordered clockwise in the logical ring. The nodes are associated with a randomly chosen unique identifier (id) represented in hexadecimal. An object in the DHT is identified by a key. A key is mapped to the node having the numerically closest id to that key. Reaching the corresponding object is made through that node. Pastry routes a message in $O(logN)$ overlay hops, N being the number of nodes in the system. Each Pastry node maintains a $O(logN)$ size routing table. Every node in the system keeps a list of

its neighbors in a *leafset* and monitors them for failure detection. A leafset is represented by the set of its $2l$ closest neighbors (the l neighbors on its right and the l neighbors on its left). The neighbors periodically exchange leafsets in order to update the information in their leafset. An example of message routing in Pastry is described in Figure 2 (a). A message is sent to a key, and routed to the numerically closest node id to that key.

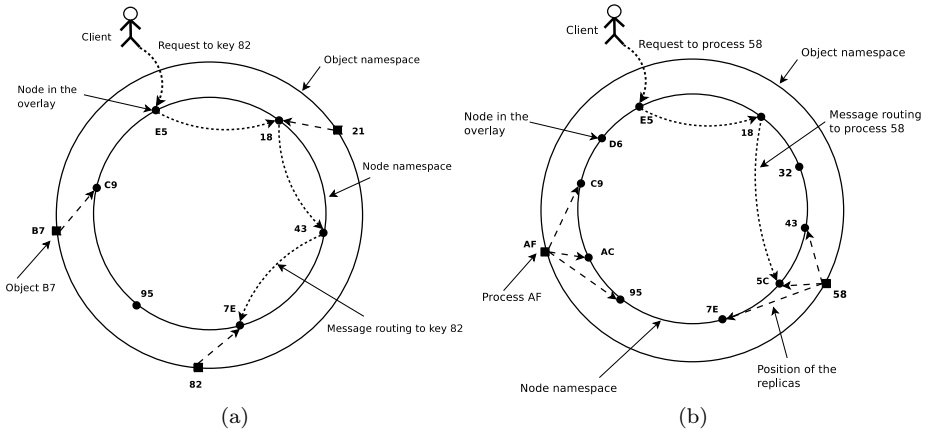


Figure 2: Message routing (a) and service active replication (b) in Pastry.

In Semias, the objects put in the DHT are services. By randomly choosing the services keys, we ensure that services are distributed over all the nodes in the system. Thanks to the DHT, a client can communicate with a service without knowing its physical position, simply by using the key that identifies the service.

2.2.3 Active Replication on Top of a DHT

By actively replicating services in a DHT, replication is made totally transparent for the clients. A client only needs to know the service key to send a request to it. It does not need to know the physical position of the service's replicas and does not need to be aware of replication. Furthermore, the DHT provides scalability and good load balancing properties for the replicated services.

Principles

To keep replication transparent to the client, we follow the natural choice of placing service's replicas on the closest nodes to the service's key[1]. Since messages in Pastry are always routed to the closest node to a service key, they always reach a node hosting a replica of the requested service, even in the event of a failure.

A detailed description of the rules that we use to select the replica positions is given in section 3.6.2. We illustrate the placement of the service replicas in Figure 2 (b). In this example, two services, respectively associated with key *58* and key *AF*, are replicated with a replication degree of *3*. If node *5C* fails,

[1]In this paper, proximity always refers to ids.

requests to service *58* will be routed to node *43* which also hosts a replica of service *58*. The node receiving a message for a service is in charge of atomically broadcasting the message to all replicas of that service.

Since node ids are chosen randomly, neighbors in the logical ring are, with a high degree of probability, uniformly distributed over the physical network. By placing the replicas of a service on neighbors in the logical network, we ensure that the risk of experiencing concurrent failures of multiple replicas is low. As we have seen in Section 2.1, there is a low risk of multiple random nodes failing together. Thus, the replication degree for a service can be small.

Possible Issues When reconfigurations in the overlay occur, replica sets have to be reconfigured accordingly. Two cases have to be carefully handled: node additions and node removals.

Node addition If a node joins the logical ring with an id closer to a service key than some of the nodes currently hosting a replica of that service, this new node is likely to receive messages for that service. One of the replicas has to be migrated so that the replicas remain on the n closest nodes to the service key. This is illustrated in Figure 3 (a). Here, a new node with id *B3* joins the overlay (step (1)). If node AC fails, then the messages sent to key AF will be received by node *B3*. A replica is migrated to node *B3* from node *C9* (steps (2) and (3)).

Node removal When a node is detected as failed, it is removed from the logical network. If the failed node was hosting a replica of a service, a new replica should be created to keep the service's replication degree constant. This is illustrated in Figure 3 (b). Here, node AC fails (step (1)) and a new replica is created on node *78* (step (2)). For now, Semias does not provide specific mechanisms for voluntary node disconnections. We handle them as failures.

Handling reconfigurations The basic solution is to reconfigure the replica sets every time a reconfiguration occurs in the peer to peer overlay. However, creating or migrating a replica involves a state transfer which can be costly, especially in a wide area network. Furthermore, in a dynamic environment, modifying the groups on every reconfiguration from the peer-to-peer overlay can lead to many useless reconfigurations. In Figure 3 (a) for instance, when a new node joins the system with the id *B3*, the replica set of service AF should be modified: the replica hosted by the node *C9* should be migrated to the new node *B3*. If immediately after this reconfiguration, the node AC fails, a new replica will have to be created again on the node *C9*, to maintain a constant degree of replication for the service.

To avoid useless state transfers, we want Semias to delay non-critical reconfigurations and to make use of this time to gather information about the nodes in the system to take better reconfiguration decisions.

2.3 Related Work

In this section, we detail the related work. We first study high availability through service replication in existing grid middlewares. Then we briefly present

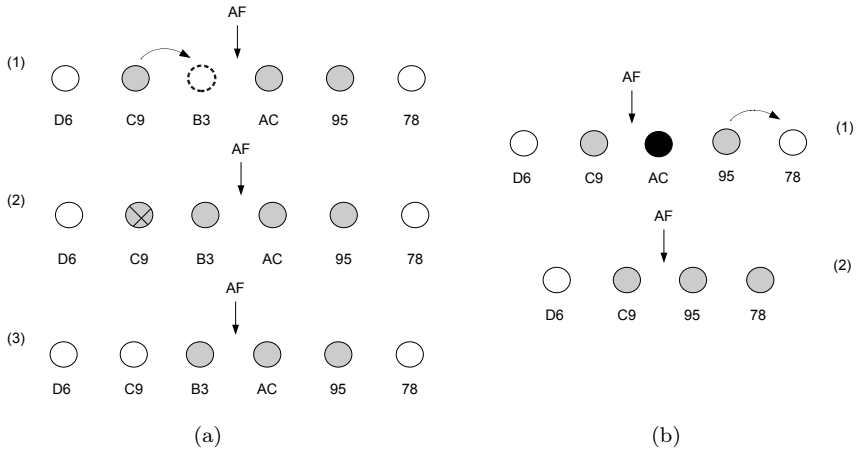


Figure 3: Steps that should be taken to reconfigure the replica set due to changes in the overlay. In (a) the replica set is changed when a new node with the id closer to a key than some replicas arrives. In (b) the replica set is changed when a node hosting one replica fails. Grey nodes are hosting a replica. The black node is the failed replica and the dashed one is the newly arrived node. The state transfer between the replicas is represented by an arrow.

the characteristics of dynamic group communication systems for active replication. Finally, we present works on active replication on top of a structured peer-to-peer overlay that address the problem of reconfigurations.

2.3.1 Services Replication in Grids

Few works have addressed the problem of grid services availability. In Migol [29] active replication is implemented for one of its core services. The replicas are distributed on different sites. Two different implementations are proposed. The first one is using a token-based protocol to provide the total ordering of the messages. The second protocol is sequencer based. They use timeouts to remove the failed replicas from the group. However, they do not mention if or how they maintain the replication degree.

XtreemOS proposes to combine active replication and mobile IPv6 [30] to make replication of services transparent for clients. This solution does not handle the problem of selecting replicas position and so, does not handle load balancing.

Zhang et. al [31] propose passive replication for non deterministic services. They use a heart-beat mechanism to detect the failure of the primary, which is changed whenever is detected as failed. Wrong suspicions are costly, because they imply a useless change of the primary. In wide-area networks failure detection is unreliable and there could be many wrong suspicions, so their approach is more suitable for clusters. In XtremWeb, clients submit jobs to a coordination service which schedules them on a set of participating workers. The coordina-

tion service is important for the system and passive replication is used to make it available despite failures [32].

Other grid middlewares use replication to improve the reliability of the executed jobs [33, 34]. Vishwa [33] and Zorilla [34] are grid middlewares based on peer-to-peer overlays. Zorilla is using a structured peer-to-peer overlay for decentralized scheduling of jobs. Each job is associated with a distributed job object that contains the state of the job and its files. The job objects are replicated using a technique close to passive replication. Vishwa is storing the information related to the tasks in a DHT and replicates it over multiple nodes. However, they do not describe the replication technique that is used. In both cases, it is not specified if and how the replication degree is kept constant, so we assume they do not provide self-healing for their groups of replicas.

2.3.2 Active Replication in Dynamic Systems

To implement active replication a group communication system is required. Group communication systems provide group membership and group communication primitives, like atomic broadcast. There are two types of architectures, based on the relation between group membership and atomic broadcast. The first type implements atomic broadcast by using group membership [35]. Many atomic broadcast algorithms require a perfect failure detector, i.e. a failure detector that does not make mistakes. This type of failure detector can be obtained by using the group membership: whenever a group member is suspected to have failed, the suspected member is removed from the group through a membership change. Thus, the atomic broadcast is implemented on top of the group membership. In this case, wrong suspicions have an important impact upon the group because they lead to many membership changes.

The second type of architecture implements group membership by using atomic broadcast [18]. The atomic broadcast is based on a consensus algorithm which can use an unreliable failure detector [36], i.e. a failure detector that can make mistakes by wrongly suspecting non-failed processes or not suspecting failed processes. In this way, the suspicion of a process does not lead necessarely to its removal from the group. This architecture dissociates the suspicions from node evictions and allows making the decision of removing the process separate from group membership. This is advantageous in dynamic systems where there could be many wrong suspicions. Also, it allows the implementation of multiple reconfiguration strategies that could cope with the dynamicity of the system. These advantages lead us to choose it as a base for Semias.

2.3.3 Active Replication on Top of a Peer-to-Peer Overlay

PaxonDHT [37] proposes a solution for implementing the Paxos [38] consensus algorithm on top of the Pastry DHT as a basic block for active replication. However, PaxonDHT only ensure the safety of Paxos with some probality because group membership changes are not synchronized with the run of consensus instances. A high replication degree is required to ensure safety with a high probability. In [39] the autors propose active replication as the base for building reliable peers, called virtual peers, in peer-to-peer networks. The virtual peers represent the groups of peer replicas. The replicas have fixed identifiers, therefore the reconfiguration is needed only at node failure. To ensure consis-

tency between them, they also use Paxos. However, messages sent to the virtual peer when the Paxos leader is failed and until the re-election of a new leader are lost. Therefore, a retransmision is required. They use a single timeout for detecting the failure of a node. The failed node is removed from the replica group. In wide-area networks, they could have many wrong failure detections, and therefore, many wrong reconfigurations.

Other recent works [40, 41] address the problem of implementing mutable replicated objects on top of a DHT. Thus they also need to solve the problem of handling reconfigurations. However, in their case, reconfigurations are not constrained by the *same view delivery* requirement of atomic broadcast: reconfigurations are not serialized with normal operations. In their case, the Paxos algorithm is used only to agree on the new configuration. The replicas are placed on the closest nodes to the object's key and the immediate successor of the key is designated as a *primary* [40] or a *master replica* [41]. All the operations are serialized through this node.

Etna [40] is build on the Chord DHT. It reconfigures the set of replicas whenever it notices that it is different from the set of successors from Chord, i.e. when a replica leaves or when a new node joins. The reconfigurations are not transparent to the client. Etna returns an error if there are ongoing opperations during a reconfiguration. DhtFlex [41] is designed for a generic overlay. It does the reconfiguration periodically, with the risk of loosing some replicated objects in case of many failures.

In Scalaris [42] the Paxos commit protocol is used to solve the problem of atomic commit in transactional databases built on top of a DHT. For coordonating the transactions they are using a replicated Transaction Manager and the replicas have fixed logical identifiers. The node arrivals and failures in the DHT are treated synchronously: every change in the overlay leads to a change in the replica set.

2.4 Conclusions

The dynamic nature of large scale distributed systems like grids, has a negative impact on the usability of services and the overall performance of the system. To ease the grid usage, services provided by grid middlewares should be highly available.

Structured peer-to-peer overlays are an attractive solution to provide scalability, location-independent naming and fault-tolerant routing. Some grid middlewares use them as a base for building distributed scalable services. To add high availability to such services, active replication can be used. The combination of active replication with a structured peer-to-peer overlay makes the replication transparent for the clients. The overlay offers scalability and good load-balancing properties. Two problems have to be solved to implement such a solution: keeping the replicas consistent and dealing with the reconfiguration of the groups of replicas. A few projects addressed the problem of having highly available grid services or suggested active replication on structured peer-to-peer overlays, but none of them handled properly the reconfiguration of replica groups.

In this paper, we propose a new solution, called Semias, for active replication of services on top of a structured peer-to-peer overlay. For active replication, Semias is based on an existing architecture for dynamic group com-

munications [18] that dissasociates node suspicion from node removal from the group. As this solution does not completly ensure the availability of a service in a dynamic environment, we propose a self healing protocol that takes critical reconfiguration decisions based on a predefined set of safety rules.

3 Design

In this Section we present the architecture of Semias. The architecture is composed of four main modules (layers). This makes it modular and extensible. In the next sections, we present the architecture used for the active replication of services. We discuss the choices that we made in building these modules and how they interact with each other. Finally, in Section 3.6 we describe the self healing mecanism of Semias used to ensure the high availability of replicated services despite node failures and arrivals.

3.1 Overview

The basic modules of Semias are a peer-to-peer overlay and a group communication system. This comes as a natural choice for providing active replication for services distributed over a peer-to-peer overlay. The peer-to-peer overlay provides fault-tolerant key-based routing mechanisms. The Group Communication System (GCS) implements atomic broadcast, used for active replication.

Also, the framework must be suitable for highly dynamic environments. One of its goals is to ensure the availability of the replicated services while minimizing the number of the group reconfigurations. Therefore, we added a third module in our design, called the Group Monitoring Layer (GML). The main task accomplished by the GML is to gather information about the groups that include the replicas hosted by the node and the changes in the node's neighborhood at the peer-to-peer overlay. It does so in order to take reconfiguration decisions. To ensure the service's availability, the GML applies a set of basic self-healing rules (we define these in Section 3.6.2). To get the information about the changes in the node's neighborhood, the GML receives a set of events from the overlay's maintenance algorithms.

Finally, as we wanted to make the replication transparent for the clients, we added a fourth module, called Service Communication Layer (SCL). This is used for the communication between the clients and the replicated services.

While the GCS is instantiated for every replicated service that resides on one node from the system, the other modules are represented by only one instance per node. In this way, the GCS handles the management of replicas only for the service assigned to it, while the SCL and the GML can have a view of all the groups hosted by a node. Also, the GML could be extended in the future to perform other tasks, e.g. gather statistics about the dynamicity from the node's neighborhood and adjust the self-healing rules based on that, so having it as a separate module allows to easily modify it in the future.

Figure 4 presents the interactions between the modules of our framework. When a client wants to send a request to a service, it routes a message containing this request to the service's key in the P2P overlay. The message is received by the SCL of the node on which one of the service's replicas reside. Then the SCL calls the **abcast** method provided by the GCS, to atomically broadcast

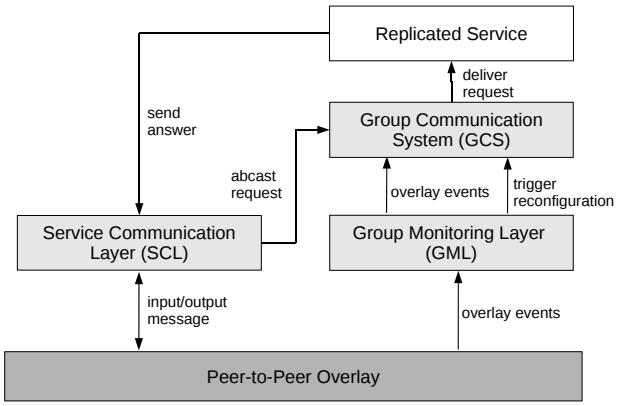


Figure 4: The interactions between the modules of Semias. The peer-to-peer overlay is a third party contribution.

the message to all the replicas. The GCS delivers the message to the service's replicas. To maintain its group configuration, the GCS receives events from the GML to reconfigure itself. The structure of every module is described in more detail in the rest of this section.

3.2 The Peer-to-Peer Overlay

We require the P2P Overlay to provide the other modules with the following functionalities:

- methods for routing a message in the overlay to a key;
- methods for direct sending a message to a neighbor from the node's leafset;
- access to the node's leafset;
- events that inform the other modules of the changes in the overlay. These events are generated at node *joining, arrival, suspicion, non-suspicion and failure*. Node suspicion must be disassociated from node failure. If such a mechanism does not exist, it could be implemented by using a separate timeout mechanism for node suspicion.

Our implementation of Semias uses an existing overlay that we slightly modified to provide the last two functionalities previously described. The events are generated by the maintenance algorithms of the overlay and are handled by the GML. How these events are generated, is presented as following:

- *Joining*: To join the system, a node A needs to contact one existing node. A asks this node to route a joining message to the key A. When this message arrives on a node that has the closest id to A, lets call it B, B generates a *joining event*.
- *Arrival*: Other nodes become aware of A and update their leafset due to the periodic leafset exchange When another node from the system, C, adds A in it's leafset it generates an *arrival event*.

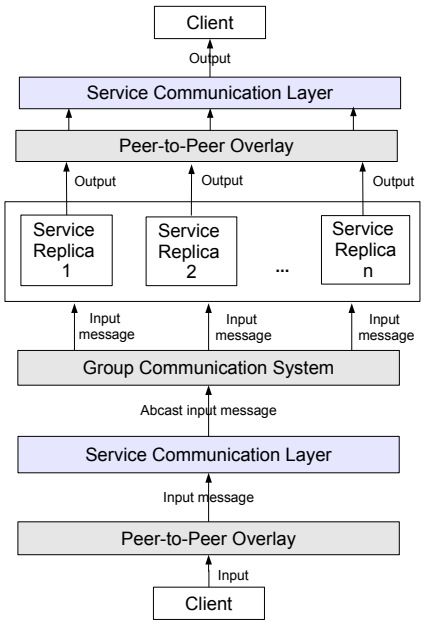


Figure 5: The interaction between SCL and the other components of the framework.

- *Suspicion*: C periodically checks if its neighbors are alive by probing them. If A does not answer before a predefined timeout, called *suspicionTimeout*, C generates a *suspicion event* and sends a second probe to A.
- *Non-suspicion*: If A answers before a second timeout, called *failureTimeout*, C generates a *non-suspicion event*.
- *Failure*: If the timeout $failureTimeout$ expired and A did not answered, C removes A from its leafset and generates a *failure event*.

Because two timeouts are used to determine if a node is failed, their values can be independently tunned. The first one can be small to quickly detect delayed nodes while the second one can be big to limit the number of wrong failure suspicions. We describe in Section 3.5 how we make use of these two timeouts.

3.3 Service Communication Layer

The SCL provides a one-to-many and many-to-one communication between a client and a replicated service. The way this module is used is presented in Figure 5.

A message sent by a client to a replicated service is routed through the overlay to the node responsible for the service key. The SCL of this node checks if the node hosts a replica for the targeted group and calls the *abcast* primitive provided by the GCS to send the message to all the group members. If the node receiving the message does not host a replica of the group (it has just joined the

overlay and has not been included in the group yet), it is not able to abcast the message. In this case, the SCL forwards the message to one current member of the group, as described in Section 3.6.

Every replica processes the client's request and outputs the same answer. We assume that the client cannot handle duplicate messages. Thus, the SCL of the node on which the client is connected delivers to the client only the first answer and rejects the others.

3.4 Group Monitoring Layer

The GML handles the events generated by the peer-to-peer overlay in order to take self-healing decisions for the groups hosted by a node. These decisions are based on a set of basic safety rules that define the minimum requirements for assuring the high availability of a replicated service. The GML also logs the failures received from the peer-to-peer overlay. This log is used by the GCS to avoid blocking during the reconfiguration process. The safety conditions and the reconfiguration process are described in Section 3.6.

3.5 Group Communication System

The GCS provides group membership management and an atomic broadcast (*abcast*) primitive. The components of GCS and their interactions are presented in Figure 6. The group membership management makes replication transparent for the services and the clients by hiding the degree of replication and the identities of the replicas. The client does not need to know the identities of service's replicas or about service's current replication degree. It can treat the group of replicas as one service. Also, the replicated service does not need to be aware of its other replicas. The abcast primitive hides the complexity of maintaining the consistency between the replicas. Thus, neither the client nor the service need to worry about replicas' consistency.

The architecture is based on the protocol stack described in [18]. The proposed solution is designed for dynamic group communication and is well modularized, allowing to modify each of its modules independently of the others. Its main characteristic is the use of atomic broadcast to build the group membership component. In this way, the membership changes are ordered with the application messages by using atomic broadcast for both.

The abcast algorithm is based on consensus. The consensus protocol requires an $\diamond S$ unreliable failure detector [36] in order to choose the coordinator of a round. This failure detector has two characteristics: (i) it ensures eventually the permanent suspicion of every failed process by every correct process; (ii) it guarantees the existence of a correct process that is eventually not suspected by any correct process.

This main feature of the framework allows dissociating node suspicion from group member eviction. Process suspicion events are used for the liveness of consensus. Process failure events are used to remove the process from the group. Thus, in the initial framework, the decision of excluding a group member could be taken separately from the group membership, from a monitoring module.

This design could take advantage of the existence of the two timeouts in the P2P overlay, *suspicionTimeout* and *failureTimeout* as mentioned in Section 3.2.

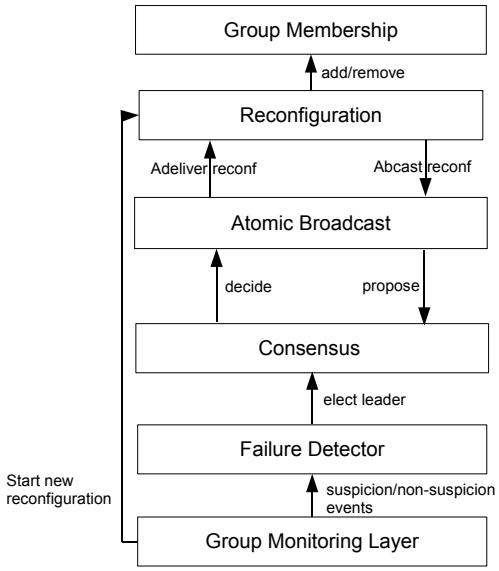


Figure 6: The protocol stack of the Group Communication System component.

These are used for notifying the suspicion of a process, and afterward, its failure. *suspicionTimeout* can be small to ensure the reactivity of the consensus algorithm, while *failureTimeout* can be large to be sure that a node is really failed before removing it. This is useful in wide area networks, where latency can be large and variable.

The GCS should be able to reconfigure itself in a dynamic environment. Adding/removing a process at every arrival/failure should be avoided, because it could lead to useless reconfigurations, as described in Section 2.2.3. A reconfiguration not only involves an abcast in the process group, which is costly, but also a state transfer to the new member of the group, so that it is consistent with the other members. So having too many useless reconfigurations could impact the performance of Semias. The decision to add or remove a process from the group is taken by a separate module, that we called *the reconfiguration component*. They are packed in a *reconfiguration message* that is ordered with the normal messages by the atomic broadcast module, and are processed when this message is delivered. The way in which the decisions are taken is described in Section 3.6.

A summary of the functionalities offered by every module is described in Table 1. The group membership offers primitives for accessing and modifying the view. The reconfiguration component provides methods for starting the reconfiguration and handling the installation of the new view. The abcast and consensus components provide well-known primitives as described in literature. The failure detector provides the leader election capability. We describe every module in the rest of this section.

Module	Functionality	Description
Group Membership	view	The current set of processes from the group
	add(p)	Adds process p to the current view
	remove(p)	Removes process p from the current view
Reconfiguration	start_new_reconfiguration()	Starts the reconfiguration process in the current view: selects and proposes the new set of processes that will be part of the group. Used when the reconfiguration process is triggered externally or by the reconfiguration component itself.
	adeliver_reconf(*message*)	Installs the new view. Called by the atomic broadcast component when a reconfiguration message is delivered.
Atomic Broadcast	abcast(type, message)	Called by the Reconfiguration component or an extern component to abcast a message. The *type* indicates if the message from the Reconfiguration component or from the application.
	adeliver(message)	Delivers the message to the application or to the Reconfiguration component.
Consensus	propose(k,proposal)	Starts the k consensus instance for this proposal. Used by the Atomic Broadcast component.
	decide(k,proposal)	Waits until a proposal is decided for the *kth* consensus instance. Used by the Atomic Broadcast component.
Failure Detector	id elect_leader()	The id of the process that is the current leader of the group. Used by the consensus component.

Table 1: Functionalities offered by every GCS component

3.5.1 Group Membership Component

The group membership component maintains the current list of group members, also called the *group view*. It offers a consistent access to the view for all the other components of the GCS and updates it by adding or removing members

when a new configuration is installed. Its role in the reconfiguration process is passive, because the decisions of adding or removing the members are taken by the reconfiguration component.

3.5.2 Reconfiguration Component

The reconfiguration component computes and installs a new group view when a reconfiguration event is triggered. To install the new configuration, the reconfiguration component of a replica sends a message to all the others members of the group containing the new view. The configuration messages are totally ordered with the normal ones, by using the atomic broadcast primitive. We describe the process of reconfiguration and the conditions that are triggering it in Section 3.6.

3.5.3 Atomic Broadcast Component

To ensure the consistency of the replicas from the group we need a uniform atomic broadcast algorithm to provide the total order delivery of messages in the group. For this, we use the uniform atomic broadcast algorithm for dynamic groups proposed by [22]. We slightly alter the initial specification in order to integrate it with the reconfiguration component. We present the pseudocode for this algorithm in Figure 7.

The algorithm is based on a sequence of consensus instances. Every instance is used to decide on a batch of messages to be delivered (lines 21-24). Every consensus instance is executed by the processes that are part of the current view when the consensus has started. Therefore, a membership change cannot affect the safety of consensus. The messages from the batch are deterministically ordered, e.g. by using a unique incremental identifier for each message, and are classified in two types:

- *Normal*: the messages that are delivered to the service.
- *Reconf*: the messages from the reconfiguration component, that will modify the group membership.

To atomically broadcast a message in the group, the replica that receives it calls the *abcast* method (line 8). After the abcast, the replica acknowledges the message at the peer-to-peer overlay. In case the message is not acknowledged, it is considered that the node receiving the message is failed and the message is re-routed in the overlay. This is done to ensure that a message sent to a replica group will be eventually received by all the group members.

To be sure that the message will be delivered by all the non-failed replicas, every replica needs to store the abcasted message in its *undelivered queue* (line 15). For doing so, when a replica receives a message for its replica group from the peer-to-peer overlay, it broadcasts the message in the group. Then, this replica waits for all the answers from the non-failed group members to be sure that they all have the message in their undelivered list (line 10). Afterwards, depending on the received answers, the replica decides if the message should be retrasmited or not (lines 11-12).

When a replica receives the broadcasted message, it replies with an *ack* or a *nack* (lines 16-19). The *ack* is sent if the replica receiving the message is in the

```
1: Variables:
2:    v    // Current set of replicas
3:    v_nb    // Current view number
4:    undelivered=∅    // List of received messages that are not delivered
5:    adelivered=∅    // List of delivered messages
6:    k=1    // The consensus instance used for deciding a batch of messages
7:
8:  abcast(type, v'_nb, msg)
9:    send message(type,v_nb,msg) to all q ∈ v
10:   wait for all answers from non-failed p ∈ v, either acks or nacks
11:   if majority of nack and type=Normal then
12:      route(message(type, v_nb, msg))
13:
14: upon receive(type, v'_nb, msg) from q for the first time
15:   undelivered ← undelivered ∪ {type, msg}
16:   if  p ∈ v and v'_nb = v_nb  then
17:      send ack(type, v'_nb, msg) to q
18:   else
19:      send nack(type, v'_nb, msg) to p
20:
21: handle decide( adeliver^k )
22:   if type = NORMAL from undelivered adeliver^k then
23:      adeliver(message)
24:   if type = RECONF from undelivered adeliver^k then
25:      adeliver_reconf(v,message)
26:   adelivered ← adelivered ∪ adeliver^k
27:
28: upon undelivered − adelivered ≠ ∅
29:   proposal ← undelivered − adelivered
30:   propose(k, v, proposal)
31:   wait until decide(k, adeliver^k)
32:   handle decide(adeliver^k)
33:   k ← k + 1
34:   pass_to_instance(k)
35:   while has next decided adeliver^k do
36:      handle decide(adeliver^k)
37:      k ← k + 1
38:      pass_to_instance(k)
```

Figure 7: Atomic broadcast algorithm for a process p.

same view as the replica that broadcasts the message. By doing so, the replica acknowledges that it has the message in its *undelivered* list and it is able to send it to any new replicas if a view change occurs. If there was a view change before the message is received, the replica sends a *nack*.

If the replica that broadcasted the message receives a majority of *acks* it means that there is a majority of replicas in the current view that have the message in their *undelivered* list and are able to initialize new replicas during a view change such that the message will be eventually delivered. Otherwise, it means that a view change took place and there is a risk that the message would not be delivered. Thus, the message is routed again in the peer-to-peer overlay,

such that it will be eventually broadcasted in the new view. However, *Reconf* messages are not routed again because they are outdated, since a view change is already occuring. This mechanism is part of our solution to avoid message loss during a replica group reconfiguration, as we proove in Section 3.7.

To ensure in a simple way that the processes from the new view will deliver the same messages from a batch as the ones from the old view, the reconfiguration messages are delivered after the application ones (lines 22-25).

Another point that should be mentioned in the description of this algorithm, is presented in lines 35-38 of Figure 7. As the Atomic Broadcast and the Consensus are separate components, a decision on some batch of messages could be made by the Consensus component, while the Abcast component is handling the current one. So, after handling the messages from the current decided instance, a check could be done, so that the next batches can be processed immediately.

3.5.4 Consensus Component

Consensus allows processes to reach a common decision despite failures. To choose an efficient consensus algorithm, we considered the behavior of representative algorithms in an environment with multiple crashes and wrong failure suspicions. These consensus algorithms are designed for the asynchronous model with unreliable failure detectors and they tolerate up to $f < n$ crash failures in a system with $2n + 1$ processes. From these, we choose the Paxos protocol. Paxos [38, 43] is efficient in environments where there are frequent wrong suspicions and multiple correlated failures [44]. However, it could be replaced by other consensus protocols.

Paxos Algorithm In the Paxos protocol, there are two types of participating processes: proposers and acceptors. The proposer proposes a value, that is accepted or rejected by the acceptors. In our implementation, we assume one process that has the role of the proposer and that is also the coordinator (leader) for the current instance. The coordinator proposes values to the set of acceptors until one is accepted. The set of acceptors is represented by all processes from the current view.

The process of trying to reach a decision is called a *round*. Only the process that is the leader can start a new round. However, processes can be wrongly suspected and there can be more than one leader at a time. The rounds execute asynchronously: not all the processes execute the same round; it depends if there are failures or if they suspect the current leader. In each round the leader tries to impose a decision value to all processes. If it succeeds then the protocol terminates, otherwise a new round is started. Thus, if there are no failures or wrong suspicions, only one round is necessary to decide a value. The execution of a round is divided in three phases:

- "Prepare" phase: the purpose is to update the leader's estimate of the decision value with the current one. If the leader failed during a round, then a new leader is chosen and it needs to know if a value was already decided or not.
- "Accept" phase: the leader sends the value that he wants to be accepted to all the other processes. If a majority of processes approve this value, than the decision is being made.

- "Decide" phase: The leader broadcasts the decided value to all the processes.

A detailed description of the algorithm, including the handling of all failure cases, is presented in Appendix A.

When deciding sequences of values, the consensus protocol is called subsequently. Each run of consensus is made in a consensus instance. For example, to agree on the $k-th$ value, the $k-th$ instance of Paxos is needed. During a run of consensus, only the processes from the view in which the instance was started participate.

3.5.5 Failure Detector

The failure detector is a module attached to each process, that gives some information about what processes have failed. Failure detectors are unreliable in asynchronous systems: they can suspect processes that have not failed or see failed processes as correct. This is because in an asynchronous system there are no timing assumptions on the delay of messages. So it is impossible to distinguish a slow process from a failed one.

The failure detector that we use is dependent on the consensus protocol. Paxos requires an Ω failure detector [45]. This failure detector provides the processes with an eventual leader election capability: it outputs only one trusted process.

To implement our failure detector, we use the Suspicion event received from the GML to build a list of non-suspected processes. Then, to select the leader, we choose the identifier of the node that is the closest to the group key from this list.

3.6 Dealing with Reconfigurations

To maintain the replication degree of a service constant and the same logical positions of the replicas despite changes in the overlay, the group configuration must be regularly updated. However, in a dynamic environment, where nodes can arrive and fail at any time, reconfiguring at every change in the overlay would be costly. Thus, Semias is periodically checking the overlay to trigger a reconfiguration if needed. Since the configuration can become invalid between two periodical checks, a set of safety conditions, described in Section 3.6.2, are verified by the GML at every node arrival or failure event. The new configuration is chosen and then proposed in the current group. If multiple configurations are proposed for the current view, only the first decided configuration is installed.

Next, we define the terms that we use when explaining the reconfiguration process. Then we outline the three safety conditions that we used to avoid invalid configurations and present the reconfiguration protocol.

Definitions For the sake of clarity, we define some terms that we use in the rest of this section.

new view: the set of nodes decided in the reconfiguration process that will replace the current one.

old view: the set of nodes that made the group view before the new configuration is installed.

new replica: a node that is added to the group when the reconfiguration occurs. Its state must be initialized to be consistent with the other replicas in the group.

old replica: a node that was included in the group before the reconfiguration occurred.

forwarding state: a special state in which the node that is not part of a group is aware of the set of group members and is able to forward messages to them.

forwarding node: a node that has the GCS component initialized in the forwarding state.

The forwarding nodes that are not included in the new view are classified in former and remaining forwarding nodes. The former ones don't respect the condition of a forwarding node in the new view that is installed. The remaining ones comply with the condition but they were not included in the new view, probably because they joined during the reconfiguration process.

3.6.1 Forwarding nodes

Since we don't change the configuration of a group at every node arrival and failure, we can reach a situation where a new node receives messages for a service but has not been included in the corresponding group of replicas. In this case, it cannot handle the message. To avoid this, when a node wants to join the overlay, the GML that receives the joining event checks if the new node is eligible to be included in an existing replica group, i.e. if its id is closer to a group's key than some of the current replicas from that group. If the condition is true, the node becomes a forwarding node.

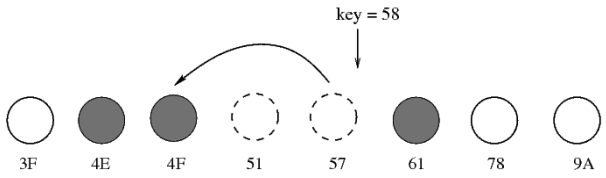


Figure 8: Example of a forwarding node that sends the messages to the closest replica. Grey nodes are hosting a replica. Dashed nodes are the forwarding nodes.

In this state, the node is able to forward any received messages to the closest node from the live replica set. Figure 8 illustrates this mechanism: 3 replicas of a service with key 58 are placed on the nodes with ids $4E$, $4F$ and 61. Two nodes with ids 51 and 57 join the system. Because they have the ids closer to the key than the current replicas, they become forwarding nodes. If messages are sent to the key 58, they will arrive on node 57 which will forward them to replica $4F$.

In order to put a node in a forwarding state, the GML sends to it an initialization message containing the current list of non-failed replicas for each of those groups. When the joining node receives the message, it initializes the GCS

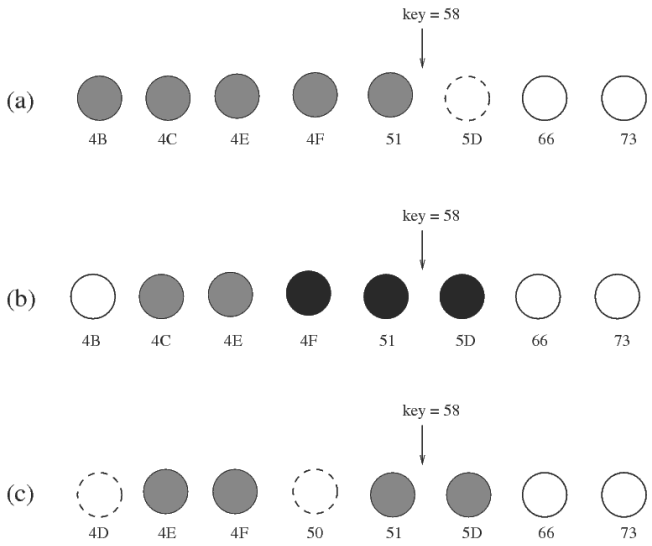


Figure 9: Cases avoided due to safety conditions. Grey nodes are hosting a replica. Black nodes represent the failed replicas. Dashed nodes are the newly arrived nodes.

state for those groups. At this point, the node is ready to announce its presence in the overlay.

Forwarding nodes mechanism requires to modify the rule for placing the replicas in the DHT: we select the closest node on both side of the key and *n-2* other closest nodes to the key. This is illustrated by Figure 9 *(a)*. In this Figure, replicas are placed on the 5 closest nodes to the key 58: $4B, 4C, 4E, 4F$ and 51. If a new node arrives with id *5D*, it should be a forwarding node because it becomes the closest node to the key. But to join, it will contact node *66*, which is the closest node id to *5D*. Node *66* could not make *5D* a forwarding node because it is not aware of group *58*. Therefore, node *66* has to be included in the set of replicas.

3.6.2 Safety Conditions

Checking for new configurations periodically could lead to situations where the current configuration of a group becomes invalid. To avoid this problem, every member of the group can propose a new configuration when it notices that the current one is about to become invalid. The GML is in charge of detecting critical configurations and triggering a reconfiguration. We define three safety conditions that the GML checks at every failure and arrival event. They are illustrated in Figure 9. In this example, a service has a replication degree of five and is associated with key *58*.

Condition 1: *There must always be a majority of non-failed replicas.*

We set the failure limit at n where the size of the group is equal to $2n + 1$. This limit represents the number of failures that the consensus protocol can

tolerate. In the example from Figure 9 *(b)*, this limit is broken because the majority of replicas fails.

Condition 2: *There must always be one replica on each side of the group key.*
This condition is required by the forwarding node mechanism, as described in Section 3.6.1.

Condition 3: *A replica must have in its leafset all the nodes hosting the replicas from its group.*
Every replica must be able to receive notifications for all the replicas of its group, in order to detect the suspected and the failed group members. But a node receives failure notifications only for the nodes that are in its leafset. If there are multiple node arrivals, old nodes could be removed from the leafset. In the example from Figure 9 *(c)*, we assume that the leafset of node *51* is the one presented. The arrival of nodes *4D* and *50* leads to the removal of node *4C* from the leafset of node *51*. To avoid this, we must install a new configuration before any replica is removed from the leafset.

When deciding these safety conditions, we assumed that reconfigurations are fast enough so that the risk of having a node arrival or failure impacting a group while a safety condition is not valid is very low. However, in highly dynamic systems, where these assumptions would not hold, the safety conditions would have to be strengthened.

3.6.3 The Reconfiguration Process

Because the reconfigurations in the overlay are not synchronized with the group reconfigurations, the current configuration of a replica group is replaced with a new one through a reconfiguration process. This process allows the change of multiple replicas in one reconfiguration with no message loss. We describe next the process of reconfiguring the replica group and we give a proof of how a replica configuration is successfuly changed, with no message loss.

The Reconfiguration Protocol

The reconfiguration process is divided in three phases: (i) configuration proposal; (ii) configuration installation; (iii) initialization of new replicas. During the reconfiguration process, the GML of the participating nodes keeps monitoring the failure events to avoid a deadlock in the process. A complete change of all the group members between two views is allowed.

The exchanges of messages between the new and the old replicas during this process is illustrated in Figure 10. In this example, replicas R_1, R_2 and R_3 belong to the old view and replicas R_2, R_3 and R_{new} compose the new view. The configuration is proposed by R_2 and abcasted to the other replicas. The installation of the new view occurs when the message is handled by the replicas. R_{new} is added to the group and it's state is initialized. R_1 is removed from the group and its GCS state is cleared.

The types of messages exchanged during the protocol are the following:

- $Reconf(v_{nb}, v_{new})$ - Abcasted by a replica when it wants to propose a new view. It contains the current view number and the new view that will be

installed. Because multiple replicas could propose a new view change, the current view number is used to install only the first new view proposed.

- *Invite_New_Replica*($v_{nb}, v_{old}, v_{new}, k, adelivered, undelivered$) - Sent by every old replica to a new replica. It is used to initialize the new replicas in the current state in which all the other replicas are. It contains: (i) the current view number used to make a distinction between the first received message and the others, v_{nb}; (ii) the old view, v_{old}, needed to select an old replica to ask the state to; (iii) the new view, v_{new}, required to initialize the group membership; (iv) the current instance of consensus,k, the *undelivered* and *already delivered* messages, required to initialize the state of the abcast component.

- *Ask_Replica_State*(n_v) - After a new replica is included in the current view, it sends this message to an old replica to retrieve the current replica state.

- *Send_Replica_State*(*replica_state*) - Sent by an old replica in order make the new replica consistent with the current ones.

- *Reconf_Ack*() - The new replica announces the successful initialization of its state.

- *Destroy_FwNode*() - Sent by the old replica to the former forwarding nodes to ask them to revert to their normal state.

- *Update_FwNode*(v_{new}) - Sent by the old replica to the remaining forwarding nodes to update their current list of non-failed replicas. The remaining forwarding nodes need this information to forward the messages to the replicas from the new view until the next configuration is installed.

We present the pseudocode for the reconfiguration protocol in Figures 11 and 12. The phases of the protocol are detailed in the rest of this section.

Phase 1: Configuration proposal The process starts when one of the replicas of the current view proposes the next configuration. The reconfiguration component computes the new view from the information provided by the GML. Then it packs the proposal in a *Reconf* message and abcasts it in the group. Several replicas can abcast a proposed new view, but only the first delivered proposal is taken into account.

Phase 2: Configuration installation Every old replica handles the configuration message (lines 8-16, Figure 11) and makes its view change. In this phase no other application messages are processed because the *Reconf* message is ordered with the normal ones. Application messages are buffered until the new configuration is installed. When changing the view, every replica from the current view saves the current state of the service and then updates its current view by adding new group members and removing the ones that are not part of the new view. Then, every old replica sends to the new replicas an *Invite_New_Replica* message with the information required to initialize the GCS state.

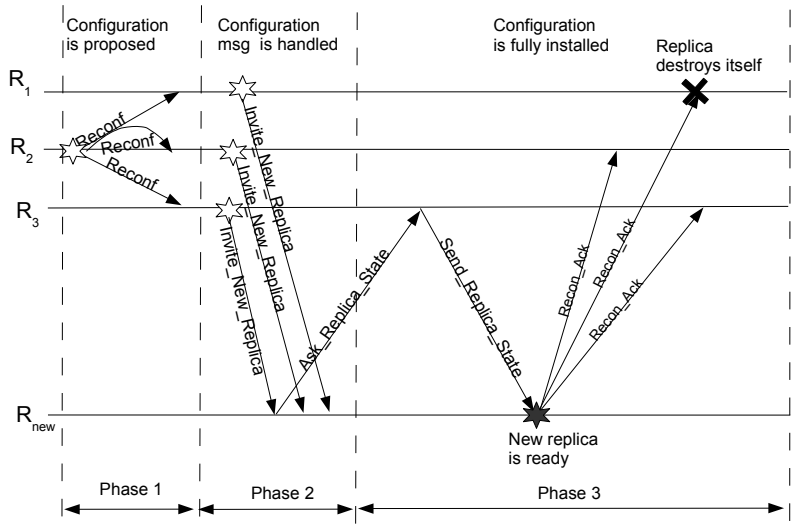


Figure 10: Message exchanges between the old and the new replicas during a normal reconfiguration process.

The installation of the new view finishes with the clearing and the updating of the remaining forwarding nodes. Every old replica sends a *Destroy_FwNode* message to the former forwarding nodes and an *Update_FwNode* to the remaining forwarding nodes.

Phase 3: Initialization of the new replicas When a node receives the first *Invite_New_Replica* message, a GCS state for the new replica is created and initialized with the information provided (lines 7-11, Figure 12). For every other *Invite_New_Replica* that is received, the new replica updates its GCS state with the new information, if any (lines 7-11, Figure 12). If the node was a *forwarding* node, it stops sending the messages to the old replicas after its GCS state initialization, because it is now able to abcast them. Also, at this point the new replica starts receiving the messages abcasted in the group.

To complete its initialization, the new node must get the current state of the replicated service from one of its replicas. To balance the state transfers, every new replica sends an *Ask_Replica_State* message to the non-failed old replica with the id closest to itself (line 11, Figure 12). If the answer is not received before a timeout, probably because the old replica crashed, the new replica sends the request to a different old replica.

When an old replica receives an *Ask_Replica_State* message, it sends a *Send_Replica_State* message containing the saved service state (line 22, Figure 11). If the state was not saved, meaning that the old replica is delayed and hasn't delivered the *Reconf* message yet, it delays the answer until it saves the replica state. After the new replica installs the state, it sends a *Reconf_Ack* message to the replicas from the old view, announcing them that its initialization was successful (lines 16-17, Figure 12).

```
1: Variables:
2:    v    // Set of processes(replicas) in the current view
3:    v_old ← ∅    // Set of replicas of the old view
4:    replica_state    // Saved replica state
5:    remaining_fw    // Set of remaining forwarding nodes
6:    old_fw    // Set of old forwarding nodes
7:
8: adeliver_reconf( v, message(Reconf, v_nb, v_new))
9:    save current replica state in replica_state
10:   v_old ← v
11:   Compute remaining_fw
12:   Compute old_fw
13:   v ← v_new
14:   send message(Invite_New_Replica, v_nb, v_old, v_new, k, adelivered,
      undelivered) to all p ∈ v − v_old
15:   send message(Update_FwNode, v_new) to remaining_fw
16:   send message(Destroy_FwNode) to old_fw
17:
18: upon receive message(Ask_Replica_State, n_v) from q
19:   if replica_state is not available then
20:      Delay answer to q until replica_state is saved
21:   else
22:      send message(Send_Replica_State, replica_state) to q
23:
24: upon receive message(Reconf_Ack)
25:   if all Reconf_Ack messages from non-failed p ∈ v − v_old are received
      and p ∉ v then
26:      Destroy replica
27:   else
28:      Delete replica_state
```

Figure 11: Reconfiguration protocol for an old replica **p**.

When all the *Reconf_ Ack* messages from non-failed new replicas are received (line 25, Figure 11), the old replica can discard the saved replica state (line 28, Figure 11) and, it can also be destroyed if it is not in the new view (line 26, Figure 11), because the new replicas are initialized and will not ask for the state anymore.

3.7 Proof of the reconfiguration protocol

In this section we proove that as long as there is a non-failed majority of replicas during the reconfiguration process, the protocol presented in the previous section allows replacing all the replicas between two views while avoiding message loss during the view change.

Theorem 1 *If there is a majority of live replicas during the reconfiguration process, all replicas can be replaced in a single view change.*

Proof Since we have a majority of non-failed replicas, the abcast algorithm (Figure 7) ensures that the *Reconf* message is delivered by all the non-failed

```
1: Variables:
2:    v    // Current set of replicas
3:    v_old ← ∅    // Set of replicas from the old view
4:    undelivered ← ∅    // list of undelivered messages from abcast
5:
6: upon receive message(Invite_New_Replica, v_nb v'_old, v_new, k,
   adelivered', undelivered')
7:    if message received for the first time then
8:       Initialize GCS state(k, adelivered', undelivered')
9:       v_old ← v'_old
10:      v ← v_new
11:      send message(Ask_Replica_State) to closest non-failed p ∈ v_old
12:   else
13:      undelivered ← undelivered ∪ undelivered'
14:
15: upon receive message(Send_Replica_State, old_replica_state)
16:   Initialize replica with old_replica_state
17:   Send msg(Reconf_Ack) to all p ∈ v_old
18:
```

Figure 12: Reconfiguration protocol on a new replica.

replicas and a reconfiguration process occurs on each of them.[2] During the reconfiguration, each of these replicas sends an *Invite_New_Replica* message to the new replicas (line 14, Figure 11) to initialize them. Since there is a majority of live replicas that send the *Invite_New_Replica* message, all the nodes that will have to create new replicas will eventually receive at least one of these messages and will initialize the new replicas.

For the new replicas to get the current replica state, there must be one live replica that is able to send it to them. This is less than our condition of having a live majority, and thus we can be sure that every new replica can finish its initialization. After the new replicas receive the state, they are initialized and any old replica that is not part of the new view can be destroyed (lines 25-26, Figure 11). Thus, the reconfiguration protocol does not require the new view to contain any of the old replicas and we can conclude that all the replicas can be replaced during one view change.

Lemma 1 *Every message received by a replica or a forwarding node will be eventually abcasted in the current group.*

Sketch of Proof Let's assume that a message is sent to a replica group using the routing functionality of the peer-to-peer overlay. If all the replicas from the current group are hosted by the closest nodes to the key, then the message will be received by one live replica. Otherwise, the message will be received by a forwarding node. We have to prove that in both cases the message will be abcasted in the current replica group.

[2]As the abcast algorithm is based on consensus, a majority of non-failed processes is required for the consensus to reach a decision.

Case 1 If the message is received by a live replica, then the SCL from the node hosting the replica will call the *abcast* method (line 8, Figure 7). This ensures that the message is abcasted in the group.

Case 2 If the message is received by a forwarding node, then the SCL from this node will send the message to one replica from its current list of replicas. If the forwarding node is up-to-date, meaning that it has the list of replicas from the current replica group, it sends the message to one live replica. This replica will eventually receive the message and will abcast it in the group.

Otherwise, we assume that a view change happened and this forwarding node hasn't received the *Update_FwNode* message so it has the list of old replicas. Thus, it forwards the message to one of the old replicas. For this situation we identify two sub-cases: i) the message is received by a node hosting an old replica that is part of the current view; ii) the message is received by a node that does not hosts a replica of the current view. In the case (i) the replica abcasts the message in current view. In the case (ii) the node receiving the message routes it again in the peer-to-peer overlay. Thus, the message can be received again by a forwarding node that is not up-to-date and the process will be indefinetly repeated. Therefore, for this case we need to assume that there will be eventually a period of stability in the neighborhood of the service's key during a reconfiguration process, in which no new nodes that comply with the condition of a forwarding node will arrive. This period is enough to ensure that the forwarding nodes will be either updated either included in the replica group. And so, in both situations the message will be received by one live replica, which will abcast it in the group.

Lemma 2 *Any abcasted message will be eventually in the undelivered queue of every active replica.*

Proof We need to prove that if a replica abcasts a message by calling the *abcast* method from Figure 7 then the message will be in the undelivered queue of every replica from a view that at one point will become the current one. We consider three cases: (i) the replica abcasting the message fails during the abcast; (ii) the replica abcasting the message does not fail and there is no view change during the abcast; (iii) the replica abcasting the message does not fail but there is a view change during the abcast.

Case (i) Because the replica fails before finishing the abcast and the acknowledgment at the peer-to-peer overlay is done after the abcast, the message is not acknowledged. In this case, the message is re-routed and it will be eventually received by a replica. Then, from Lemma 1 we conclude that the message will be eventually abcasted in the group.

Case (ii) If the replica receiving the message does not fail and there is also no view change during the abcast, every replica will eventually receive the message. Thus, every replica will have it in its undelivered queue (Line 15, Figure 7).

Case (iii) For this case, we prove that the new replicas will also have the undelivered message in their undelivered queues. As the view change occurs during the abcast, not all the replicas receiving the message are in the same view. This leads to two situations in which we need to prove that the new replicas will have the undelivered messages.

The first situation is when there is a majority of replicas that has not changed the view. These replicas send an *ack* to the replica that abcasts the message

(line 17 Figure 7) and put the message in their undelivered queues. Afterwards, when they make the view change, they pack the undelivered message in the *Invite_New_Replica* message and send it to the new replicas (line 14 Figure 11). By having a majority of replicas that send the *Invite_New_Replica* message, and from our assumption that there is a majority of non-failed replicas during the view change, we can conclude that there will be at least one live replica that sent the undelivered message to the new replicas. Thus, all of them will eventually have it in their undelivered queues.

The second situation is when there is a majority of replicas that have changed the view. These replicas send an *nack* to the replica that abcasts the message (line 19 Figure 7). They have already sent an *Invite_New_Replica* to all new replicas and so they will not send the undelivered message to them. Thus, when the replica abcasting the message receives the majority of *nack* it routes the message in the overlay (line 12 Figure 7). This ensures that the message will eventually be received by one replica from the current view. As this replica will abcast it in the current group, all replicas, including the new ones, will eventually have it in their undelivered queues.

As we showed how every replica, including the new ones will eventually have the message in their undelivered queue, we can now conclude that every active replica, i.e. that is part of a view that at one point will become the current one, will eventually have the abcasted message in its undelivered queue.

Theorem 2 A message sent to a group of replicas will be eventually delivered to the replicated process.

Proof According to Lemma 1, any message sent to a replica group will be eventually abcasted in the group. Then, by using Lemma 2, we can conclude that every abcasted message will be in the undelivered queue of every replica from one view. At this point, the consensus leader will eventually propose the message, and so it will be eventually delivered to the replicated process.

3.8 Conclusions

In this Section we presented the detailed design of our solution for implementing active replication for services on top of a DHT. The main characteristics of our framework are the following. First, we use a group communication architecture that places group membership on top of an atomic broadcast based on consensus. Node suspicion is decoupled from node failure. The suspicion event is used for consensus and the failure event for reconfiguring the group of replicas. We added to the initial architecture a reconfiguration protocol that tries to balance the transfer of the replica states when changing the group configuration. The complete initialization of the new replicas is done after the view change, so that the old replicas can continue processing messages without waiting for the new replicas. Then, we placed the responsibility of avoiding critical configurations in a separate module, that monitors the changes in the overlay. The basic mechanism that we used to ensure the self-healing property of the replica groups is based on a periodic check of the group configuration. To ensure that the current group configuration remains valid between two checks, a set of basic safety rules is used. Thus, we reduce the number of reconfigurations by avoiding to make a reconfiguration decision for every change in the overlay.

4 Implementation in Vigne

In this Section, we first present the implementation of Semias prototype. Then we describe what has to be done to execute a service using Semias. Finally, we present how we used Semias in Vigne.

4.1 Semias Prototype

To be compatible with Vigne's implementation, we implemented Semias as a C library. The framework is run as a part of the Vigne daemon on every node of the system. To join the grid, a node first has to start Semias and join the overlay. In the current prototype, the replication degree for services is constant, so it must be chosen when Semias is started on the first node. The GML and SCL are initialized during Vigne initialization stage. For every service replica a GCS instance is initialized, during the initialization stage of the service. The life-time of a GCS instance is given by the life-time of the replicated service.

For the Semias overlay we use the Pastry implementation of Vigne. The routing algorithms of the overlay are slightly modified to take into account suspected nodes: a node always tries to forward a message to the most appropriate non-suspected node. Furthermore, to improve performance, a request doesn't always need to reach the node with the closest id to the key: if a routed request reaches a node hosting a replica of the targeted service, the SCL of that node directly *abcasts* the message in the group.

As described in Section 3.5, for the consensus component we implemented a basic Paxos algorithm. Consensus instances are run sequentially. However, the modularity of the framework allows improving the consensus component without affecting the other components. Running multiple instances in parallel or using Fast Paxos [46] could be some future solutions to improve performance.

4.1.1 Replicating Services Using Semias

To run an existing service in Semias, a few modifications of the service have to be performed. First, functions to save and load the state of the service must be implemented. Then, when the service is started, it is registered in Semias, so that a GCS structure can be created for it. Also, when the service is stopped, it should be un-registered from Semias. The communication primitives used by the service have to be replaced with the functions provided by the SCL. Now, to send a message, the service must use its associated key to identify itself and send the message through the overlay. Also, the message is delivered to it by the GCS and it's not received directly from the client. Finally, the service's clients have to be modified to address a service using its associated key instead of a physical address. A client communicates with the local Semias daemon, which routes the requests to the targeted service.

4.2 Replicating the Application Managers

We used Semias to make the application manager self-healing and highly available. The application management is the key service of Vigne since it is in charge of the applications during their life-cycle on the grid. If the node on

which the application manager resides fails, then the client will not receive its results or know its application's status.

To replicate the application manager, we made the following modifications. We implemented the functions to load and save the state of the application manager. This state contains information about the selected resources on which the aplication runs and the state of every task from the application. Then, we modified the communication methods used by the application manager, such that all the communications are done through SCL. And, finally, as the client already communicates with the application manager through the P2P overlay, no modification was required for it.

4.3 Conclusions

By using Semias we made the application management service of Vigne highly available. Implementing Semias in Vigne lead us to believe that combining active replication and peer-to-peer overlays is a good way to make replication transparent for the user. Semias is quite generic, so it could be seen as a stand-alone framework, and we believe that, with some effort, it could be used independently of Vigne.

5 Evaluation

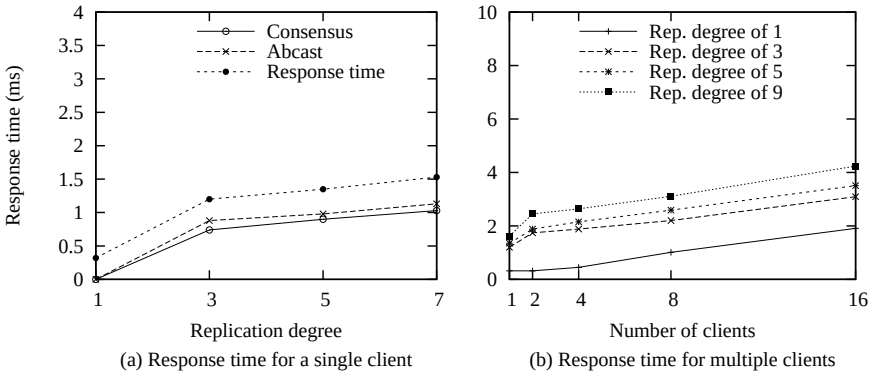
In this Section we present the evaluation of the replicated service and the behavior of Semias in the context of node arrivals and failures. To see the impact of replication on the performance of a service, we measured the response time and throughput of the replicated service in both a static and dynamic environment. For these experiments, replica positions are chosen manually from different sites on PlanetLab or Grid'5000, to highlight the characteristics of Semias. Finally, we measure the total number of reconfigurations done when Semias is used to replicate multiple applications managers in a dynamic distributed system.

5.1 Evaluation in a Static Environment

To measure the average response time and throughput of the replicated service, we ran experiments with 1 to 7 replicas placed on the same cluster and then on different sites from Grid'5000 and PlanetLab. First, we measured the average response time of a client's request. Additionally, we measured the time to abcast the request in the group of replicas and the time needed to execute the corresponding consensus instance.

The response time for a request is composed of the time needed to send the request and receive the answer, plus the time to abcast the request in the group of replicas. The time to abcast is measured as the difference between the moment the leader receives the message in the undelivered buffer and the moment the message is delivered. The time for executing the consensus instance represents the time for reaching an agreement on the proposed batch of messages.

For the experiments ran on the same cluster we chose the Rennes site. Each node has a Intel Xeon processor at 2.33 GHz and 4GB memory. The nodes are connected through a Gigabit Ethernet network with an average latency of $20\mu s$.



(a) Response time for a single client

(b) Response time for multiple clients

Figure 13: Application manager's response time on a cluster for different replication degrees.

Figure 13(a) shows the average response time of one request for multiple replication degrees. For a replication degree of 1 the average response time of a request is around $0.32ms$ while for a replication degree of 3 it becomes $1.2ms$, increasing almost four times. This overhead is the time required by the consensus to decide the delivery order of the received messages. The response time increases gradually with the replication degree because the leader always needs to wait for a majority before reaching the decision.

In Figure 13(b) we present the average response time of the application manager when multiple clients issue requests at the same time. The increase in response time for a replicated service is due to the fact that we don't run multiple instances of consensus in parallel. Therefore, the received messages are decided after the previous instance finishes. Since we cannot assume that requests from different clients arrive at the same time and are proposed in the same batch, multiple instances of consensus could be required to decide them.

Figure 14 shows the response time on Grid'5000 and PlanetLab, for various replication degrees. On Grid'5000 replicas were placed on Grenoble, Lyon, Sophia, Bordeaux, Orsay and Nancy sites. The leader from the consensus was located on a node from Grenoble site. The client issued requests from a node from Lille. On PlanetLab replicas were placed in Albany, Toronto, Hannover, Auckland, London, Delft and Sophia-Antipolis. The leader was located in Auckland and the client in Montreal.

The performance drop between the non-replicated application manager and the three times replicated application manager is due to the high network latency between the site on which the leader is located and the sites hosting the other replicas. Even when the replication degree is increasing, Paxos average latency has a constant value. This is because the consensus leader only waits for the faster majority in order to reach a decision.

Figure 15 shows the average throughput of a replicated service on Grid'5000 and PlanetLab in two different situations. In the first situation we simulated a service that is doing some computation (aproximating the value of pi) before sending back the answer to the client. The time required for the computation is

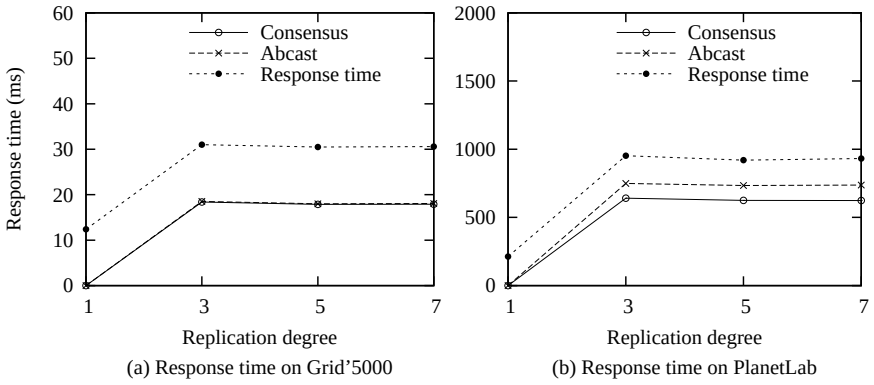


Figure 14: Response time of the application manager on Grid'5000 and on PlanetLab for different replication degrees.

higher than the time required for a consensus instance. Thus, the overhead of consensus is not noticed because the consensus instances are run almost in parallel with the computation. Moreover, as the result is received from the fastest replica, the throughput is improved. This can be better seen in the experiment ran on Grid'5000. Here the throughput of the replicated process is more than two times the throughput of the normal process. In the second situation, we sent requests to get the state of the application. Every request involved the service asking the nodes on which the application is running for the application state. As no computation is done, but only message processing, we can notice the overhead brought by the high network latency and the time required for consensus to decide a batch of messages. Since we used the same replica distribution as for the evaluation of the application manager's response time, the behavior of the replicated service when the replication degree is increased remains the same.

5.2 Evaluation in a Dynamic Environment

To evaluate Semias in a dynamic context, we used a replication degree of 5 and put the application manager replicas on 4 different sites. The client was located on a node from Lille and sent a request every 100 ms. The configuration of the group was periodically checked by Semias every 300 seconds. SuspicionTimeout was set to $3sec$ and failureTimeout to $60sec$. We display the average response time per second in Figure 16.

5.2.1 Impact of Node Failures

In order to measure the impact of successive failures in the group of replicas, we manually crashed four nodes that were hosting a replica at an interval of 80 seconds each. Figure 16(a) shows the variation of the response time until all the failed replicas are replaced. The failure events and the distribution of the replicas after every reconfiguration are summarized in Table 2.

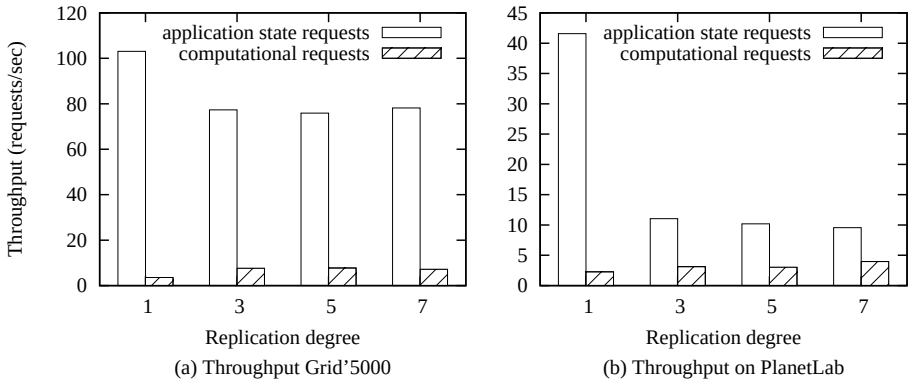


Figure 15: Throughput on PlanetLab and Grid'5000 for different replication degrees.

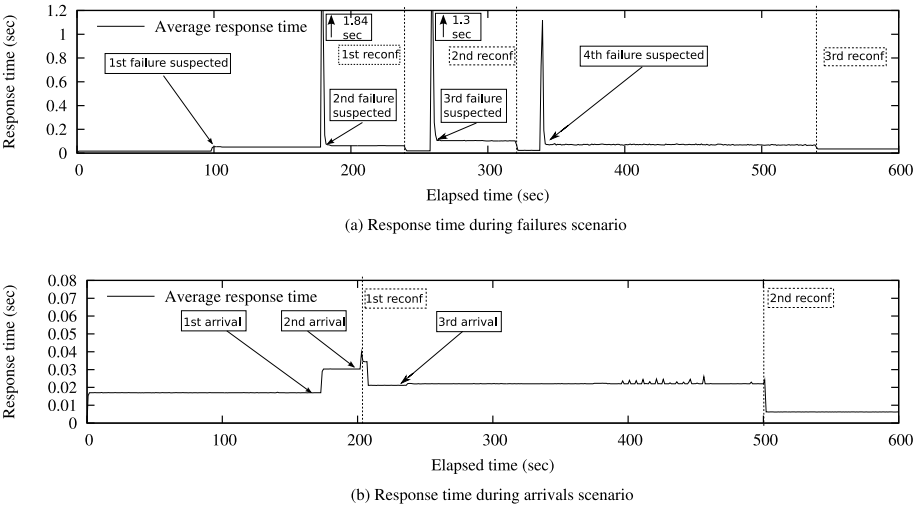


Figure 16: The response time of the application manager in the context of node failures and arrivals

The failed replicas are replaced gradually, during three reconfigurations. The first and the third reconfiguration occur at the periodic checking of the group view. The period of reconfiguration is measured from the time the service is started. Therefore, the first check occurs before 300 seconds. When the first reconfiguration occurs, the second failure is not yet detected and the failed replica is still included in the group. Therefore, after the third failure is reported, the second reconfiguration occurs, due to safety condition 1.

We notice that every failure increases the response time of the replicated service. There are three causes that can lead to this: (i) the time required to elect a new leader for consensus and to route messages to it; (ii) the positions

Reconf	R_1	R_2	R_3	R_4	R_5
0_0	*Lyon*	*Orsay*	**Orsay**	Nancy	Bordeaux
0_1	*Lyon*	*Orsay*	**Orsay**	Nancy	Bordeaux
0_2	*Lyon*	**Orsay**	Orsay	Nancy	*Bordeaux*
1_0	*Lyon*	**Orsay**	Orsay	Bordeaux	*Lyon*
1_1	*Lyon*	Orsay	Orsay	**Bordeaux**	*Lyon*
2_0	*Bordeaux*	*Lyon*	**Bordeaux**	*Lyon*	Orsay
2_1	Bordeaux	**Lyon**	Bordeaux	*Lyon*	*Orsay*
3_0	Nancy	Bordeaux	**Lyon**	*Lyon*	*Orsay*

Table 2: Replica distribution after each failure/reconfiguration event, starting from the initial configuration. The majority that could give the answers during a consensus run is shown in italic and the consensus leader is shown in bold. The failed group members are marked as grey.

of the non-failed replicas that can answer to the leader during consensus; (iii) the management of the failed links in the overlay. This overhead is reduced at every reconfiguration.

The impact of crashing the replica closest to the service key can be noticed for the next three failures. Every failure leads to a response time between 1.11 and 1.84 seconds for the messages sent at the time the node fails. This delay represents the time that it takes for the failure to be suspected. From the time the node crashes to the time the failure is suspected, the messages are routed to the failed node. In the same time, the replicas are not able to process any message because no new leader is elected for consensus. When suspicion is notified, the messages are routed to the closest non-suspected node to the key. This node also becomes the consensus leader, the replicas start to process the messages again and the response time drops to an acceptable value. The impact of the non-failed replica positions can be noticed after the third failure. Now, all the remaining live replicas represent the majority and their positions lead to a higher response time than before. The overhead of managing the failed links can be better noticed after the first failure. Since the failed replica was not included in the majority needed for consensus its position does not impact the consensus latency.

5.2.2 Impact of Node Arrivals

To measure the impact of node arrivals, we added three new nodes in the system and chose their ids such that the added nodes become forwarding nodes. We set the interval between arrivals at 30 seconds. Figure 16 (b) shows the variation of the application manager average response time until the new nodes are all included in the group. Table 3 summarizes the replica distribution for each configuration.

The presence of a forwarding node has an impact on the service's response time. This can be better noticed after the first arrival. Since this new node is in Bordeaux, the overhead is due to the network latency between Orsay and Bordeaux. The second arrival triggers the group reconfiguration because the safety condition 3 is about to be broken. The reconfiguration has an impact on

Reconf	R_1	R_2	R_3	R_4	R_5
0	*Nancy*	*Orsay*	**Orsay**	Lyon	Lyon
1	*Orsay*	Bordeaux	**Nancy**	*Orsay*	Lyon
2	*Orsay*	Bordeaux	Nancy	**Orsay**	*Orsay*

Table 3: Replica distribution during node arrivals scenario. The majority that could give the answers during a consensus run is shown in italic and the consensus leader is shown in bold. The failed group members are marked as grey.

response time. Since the new nodes are closer to the key than the old replicas, one of them becomes the consensus leader. Therefore, during a small period of time, the old replicas need to wait for the initialization of the new leader in order to process messages. Also, after the reconfiguration, client's requests are routed to one of the old replicas, until the node hosting the new replicas is taken into account in message routing. After the third arrival, the reconfiguration of the group takes place at the periodic check. Due to the positions of the faster replica majority, the new configuration leads to a better response time than before.

5.3 The Efficiency of the Self-healing Mechanisms

To see how many reconfigurations Semias is avoiding in a dynamic environment, we started 50 replicated application managers on 100 nodes of Grid'5000 distributed on 5 sites: Lyon, Lille, Nancy, Bordeaux and Sophia. The replication degree was set to 5. Node ids were chosen randomly. The reconfiguration period was set to $10mins$. In our setup, nodes arrived and failed randomly. We varied the arrival and failure period from 3 to 1 min. To evaluate the number of avoided reconfigurations, we logged the number of reconfigurations that would have occurred if the groups were reconfigured at every change in the overlay and compared it to the number of reconfigurations that really occurred. Each experiment lasted one hour and at the end all the application managers were still available.

Period of node failures and arrivals	180 s.	120 s.	60 s.
Potential group reconfigurations	97	118	220
Effective group reconfigurations	79	77	135
Reconfigurations avoided	18.5%	34.7%	38.6%

Table 4: Number of reconfigurations for various node arrival/failure rates

The results are presented in Table 4. Due to our safety conditions between 18.5 and 38.6 percent of the reconfigurations were avoided. This percentage depends on the reconfiguration period and the failure and arrival rates. In a highly dynamic system, Semias efficiently manages to reduce the number of reconfigurations and so, the amount of communication and state transfers.

5.4 Conclusions

The evaluation of the replicated application manager shows that its performance is maintained in acceptable limits. The main overhead comes from consensus. This could be improved in the future, by running multiple consensus instances in parallel or using an optimized algorithm. In a wide area network, the response time and throughput are dependent on the network latency between the replicas. Therefore, the replica placement has a bigger impact on performance than the replication degree. The behavior of Semias in a dynamic system shows that node failures and arrivals remain transparent to the client, even if they induce a delay in the service's response time. The management of several replicated application managers in a dynamic environment shows that the total number of reconfigurations is limited. This could be improved in the future by adjusting the safety conditions based on data that would be gathered about the node arrival/failure rates.

6 Conclusions

In this paper we presented a solution for providing active replication of stateful services that are distributed on top of a structured peer-to-peer overlay. Building active replication on top of a structured peer-to-peer overlay raises the issue of how to efficiently reconfigure the groups of replicas. We tried to address this issue by designing and implementing Semias. The purpose of Semias is to provide high availability for distributed services in a transparent way for the user. To our knowledge, Semias is the first implementation of reconfigurable active replication of services on top of a structured peer-to-peer overlay.

To provide active replication in a dynamic environment, we chose a group communication architecture that is suitable for such environments. This architecture allows to dissociate the suspicion of a node's failure from the node's removal from its group. This makes Semias well-suited for wide-area networks because it is not highly impacted by wrong suspicions.

We extended this architecture with a reconfiguration protocol to be able to safely handle the changes from the structured peer-to-peer overlay. Semias manages the dynamicity of the system by taking self-healing decisions. The decisions are based on a set of rules that ensure availability while minimizing the number of reconfigurations. The reconfiguration protocol balances the state transfers between the replicas and allows them to be done in parallel with the processing of messages. Moreover, the reconfiguration process can ensure that all replicas can be changed in one reconfiguration, with no message loss.

With the use of Semias, we replicated one of Vigne's key services. Semias made failures and reconfigurations of the replicated service fully transparent for the services users. The experiments run on Grid'5000 and PlanetLab show that Semias can provide a fair performance to replicated services. The self-healing mechanisms of Semias ensured the availability of services in a dynamic environment. In the same time, Semias managed to reduce the number of reconfigurations compared to the classic case, when a reconfiguration would have occurred at every node arrival or failure.

Future Work

The work presented in this paper opens up new research directions. We present now some possible future works starting with short-term ones and then describing the long-term ones.

Short term work We plan to use existing failure traces to better evaluate the reconfiguration mechanism of Semias in a realistic environment. Also, further testing Semias on grids with higher latencies between nodes will give us a clearer view of its behavior in wide area networks.

Improvements could be also done to obtain better performance for the replicated services. As we have seen from our experiments, the performance overhead is added by consensus. Running multiple instances of consensus in parallel and using an optimized version of the algorithm could reduce the overhead. Distributing the consensus instances over the participating replicas [47] has yielded better results than the classic Paxos protocol that we used, and this direction could be also investigated.

Also, the performance could be further improved by making a distinction between *read* and *write* messages. Read messages are those who do not modify the state of the service, so they do not need to be handled by all replicas. They could be handled by the closest replica to the client. Thus we could take advantage of replication to improve response time and balance the load over the replicas.

Long term work As a long term work, we plan to adapt the self-healing mechanism to the dynamicity of the environment. We want to limit the cost of reconfigurations and in the same time to keep the high availability of the services. One idea would be extend the GML such that it could monitor failure and arrival rates in the node's neighborhood and try to predict the churn rate. Then, based on these predictions, it could adjust the safety conditions and the replication degree of the service accordingly. Another idea would be to apply different reconfiguration policies based on the priority of the service. If the service is critical then maintaining a higher replication degree and having stronger safety conditions could be better. Otherwise, a smaller replication degree and relaxed safety conditions could suffice.

Finally, we believe that Semias is generic enough, so we would like to investigate the possibility of using it as a stand-alone framework and integrating it in other grid middleware, for example, like XtreemOS.

References

[1] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and I. Touche, "Grid'5000: A large scale and highly reconfigurable experimental grid testbed," *International Journal of High Performance Computing Applications*, vol. 20, no. 4, pp. 481–494, 2006.

[2] A. Iosup, M. Jan, O. Sonmez, and D. Epema, "On the dynamic resource availability in grids," in *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*, pp. 26–33, IEEE Computer Society, 2007.

[3] I. Foster, H. Kishimoto, A. Savva, D. Berry, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, J. Treadwell, and J. V. Reich, "The Open Grid Services Architecture, Version 1.5," Tech. Rep. GFD-R.80, Open Grid Forum, 2006.

[4] L. Rilling, "Vigne: Towards a Self-Healing Grid Operating System," in *Proceedings of Euro-Par 2006*, vol. 4128, (Dresden, Germany), pp. 437–447, August 2006.

[5] R. I. Resnick, "A Modern Taxonomy of High Availability." http://www.generalconcepts.com/resources/reliability/resnick/, 1996.

[6] R. Pennington, "Terascale clusters and the TeraGrid," in *Proceedings of 6th International Conference/Exhibition on High Performance Computing in Asia Pacific Region*, (Bangalore, India), pp. 407–413, 2002. Invited talk.

[7] "Folding@home," `http://folding.stanford.edu`.

[8] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "SETI@home: An Experiment in Public-Resource Computing," *Communication of the ACM*, vol. 45, no. 11, pp. 56–61, 2002.

[9] P. Eerola, B. Kónya, O. Smirnova, T. Ekelöf, M. Ellert, J. R. Hansen, J. L. Nielsen, A. Wäänänen, A. Konstantinov, J. Herrala, M. Tuisku, T. Myklebust, F. Ould-Saada, and B. Vinter, "The NorduGrid production Grid infrastructure, status and plans," in *GRID '03: Proceedings of the 4th International Workshop on Grid Computing*, (Washington, DC, USA), p. 158, IEEE Computer Society, 2003.

[10] "Enabling grids for e-science project," `http://www.eu-egee.org/`.

[11] T. Cortes, C. Franke, Y. Jégou, T. Kielmann, D. Laforenza, B. Matthews, C. Morin, L. P. Prieto, and A. Reinefeld, "XtreemOS: a Vision for a Grid Operating System," tech. rep., XtreemOS, 2008.

[12] C. Morin, "XtreemOS: A Grid Operating System Making your Computer Ready for Participating in Virtual Organizations," in *ISORC'07: Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, (Washington, DC, USA), pp. 393–402, IEEE Computer Society, 2007.

[13] I. Foster, "Globus toolkit version 4: Software for service-oriented systems," in *IFIP International Conference on Network and Parallel Computing, Springer-Verlag LNCS 3779*, pp. 2–13, 2005.

[14] E. Jeanvoine, C. Morin, and D. Leprince, "Vigne: Executing Easily and Efficiently a Wide Range of Distributed Applications in Grids," in *Proceedings of Euro-Par 2007*, (Rennes, France), pp. 394–403, 2007.

[15] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Proceedings of International Middleware Conference (Middleware 2001)* (R. Guerraoui, ed.), vol. 2218 of *Lecture Notes in Computer Science*, pp. 329–350, Springer, 2001.

[16] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz, "Handling churn in a DHT," in *Proceedings of the USENIX Annual Technical Conference*, pp. 127–140, 2004.

[17] D. Pulsipher and L. Ovoca, "Job Submission Description Language (JSDL) Specification, Version 1.0,"

[18] S. Mena, A. Schiper, and P. Wojciechowski, "A Step towards a New Generation of Group Communication Systems," in *Proceedings of International Middleware Conference (Middleware 2003)*, vol. 2672, pp. 414–432, 2003.

[19] R. Nath, "Fault tolerance of the application manager in Vigne," tech. rep., 2008.

[20] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso, "Understanding replication in databases and distributed systems," *20th International Conference on Distributed Computing Systems (ICDCS 2000)*, pp. 464–474, 2000.

[21] F. B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: a Tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

[22] A. Schiper, "Dynamic group communication," *Distributed Computing*, vol. 18, no. 5, pp. 359–374, 2006.

[23] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "The Primary-Backup Approach," *Distributed systems (2nd Edition)*, pp. 199–216, 1993.

[24] R. Guerraoui and A. Schiper, "Software-based replication for fault tolerance," *Computer*, vol. 30, no. 4, pp. 68–74, 1997.

[25] D. Powell, I. Bey, and J. Leuridan, eds., *Delta Four: A Generic Architecture for Dependable Distributed Computing*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1991.

[26] X. Défago, A. Schiper, and N. Sergent, "Semi-passive replication," in *Seventeenth IEEE Symposium on Reliable Distributed Systems, 1998. Proceedings*, pp. 43–50, 1998.

[27] M. Chérèque, D. Powell, P. Reynier, J.-L. Richier, and J. Voiron, "Active Replication in Delta-4," in *Twenty-Second International Symposium on Fault-Tolerant Computing (FTCS-22)*, pp. 28–37, 1992.

[28] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, "A survey and comparison of peer-to-peer overlay network schemes," *IEEE Communications Surveys and Tutorials*, vol. 7, pp. 72–93, 2005.

[29] A. Luckow and B. Schnor, "Service Replication in Grids: Ensuring Consistency in a Dynamic, Failure-Prone Environment," in *IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2008)*, pp. 1–7, April 2008.

[30] G. Pierre, T. Schütt, J. Domaschka, and M. Coppola, "Highly Available and Scalable Grid Services," in *Proceedings of the Third Workshop on Dependable Distributed Data Management (WDDM '09)*, pp. 18–20, 2009.

[31] X. Zhang, D. Zagorodnov, M. Hiltunen, K. Marzullo, and R. D. Schlichting, "Fault-tolerant Grid Services Using Primary-Backup: Feasibility and Performance," in *Proceedings of the 2004 IEEE International Conference on Cluster Computing (Cluster '04)*, (Washington, DC, USA), pp. 105–114, 2004.

[32] S. Djilali, T. Herault, O. Lodygensky, T. Morlier, G. Fedak, and F. Cappello, "RPC-V: Toward Fault-Tolerant RPC for Internet Connected Desktop Grids with Volatile Nodes," in *Proceedings of the ACM/IEEE Supercomputing Conference (SC2004)*, pp. 39–39, November 2004.

[33] M. V. Reddy, A. V. Srinivas, T. Gopinath, and D. Janakiram, "Vishwa: A reconfigurable P2P middleware for Grid Computations," in *Proceedings of the 2006 International Conference on Parallel Processing (ICPP '06)*, pp. 381–390, 2006.

[34] N. Drost, R. V. van Nieuwpoort, and H. Bal, "Simple Locality-Aware Coallocation in Peer-to-Peer Supercomputing," in *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID '06)*, (Washington, DC, USA), pp. 14–24, 2006.

[35] X. Défago, A. Schiper, and P. Urbán, "Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey," *ACM Computing Surveys*, vol. 36, no. 4, pp. 372–421, 2004.

[36] T. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[37] B. Temkow, A.-M. Bosneag, X. Li, and M. Brockmeyer, "PaxonDHT: Achieving Consensus in Distributed Hash Tables," in *Proceedings of the International Symposium on Applications on Internet (SAINT '06)*, pp. 236–244, 2006.

[38] L. Lamport, "The Part-Time Parliament," *ACM Transactions Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.

[39] A. Kota, U. Tatsuya, S. Masanori, I. Hayato, and M. Toshio, "Toward Fault-Tolerant P2P Systems: Constructing a Stable Virtual Peer from Multiple Unstable Peers," in *Proceedings of The First International Conference on Advances in P2P Systems (AP2PS '09)*, pp. 104–110, 2009.

[40] A. Muthitacharoen, S. Gilbert, and R. Morris., "Etna: A Fault-tolerant Algorithm for Atomic Mutable DHT Data," Tech. Rep. 993, MIT-LCS, June 2005.

[41] U. Bartlang and J. P. Muller, "DhtFlex: A Flexible Approach to Enable Efficient Atomic Data Management Tailored for Structured Peer-to-Peer Overlays," in *Proceedings of the 3rd International Conference on Internet and Web Applications and Services (ICIW '08)*, pp. 377–384, 2008.

[42] M. Moser and S. Haridi, "Atomic Commitment in Transactional DHTs," in *Proceedings of the CoreGRID Symposium, Rennes, France*, p. 151, 2007.

[43] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui, "Deconstructing Paxos," *ACM SIGACT News*, vol. 34, no. 1, pp. 47–67, 2003.

[44] P. Urban and A. Schiper, "Comparing the Performance of Two Consensus Algorithms with Centralized and Decentralized Communication Schemes," Tech. Rep. LSR-REPORT-2004-030, LSR, 2004.

[45] T. Chandra, V. Hadzilacos, and S. Toueg, "The weakest failure detector for solving consensus," *Journal of the ACM*, vol. 43, no. 4, pp. 685–722, 1996.

[46] L. Lamport, "Fast Paxos," *Distributed Computing*, vol. 19, no. 2, pp. 79–103, 2006.

[47] Y. Mao, F. Junqueira, and K. Marzullo, "Mencius: Building efficient replicated state machines for WANs," in *Proceedings of the 8th USENIX Symposium on Operating systems Design and Implementation*, 2008.

Appendix A - The Paxos algorithm

We give here a detailed description of the Paxos algorithm, that we used in Semias. We present the messages exchanged during the algorithm and how failures and delays are handled.

Outline of the algorithm The protocol is triggered when a value needs to be decided, by calling the *propose* method from an external component. Figures 17 and 18 depict the protocols on the leader and the participating process.

The leader starts a new round by sending a *Prepare* message to all the processes that participate at the current instance of consensus [3], noted with k (line 34 from Part 1 of Figure 18). When receiving such a message, a process from the group checks if it is already committed to a previous higher round by comparing the round number from the message with the round number stored locally - line 33 from Figure 17. If the process is already committed it sends a *Reject* message to inform the leader. Otherwise, it sends a *Response_Prepare* message with the value already accepted for the current instance, together with its round number - if any (line 37 Figure 17). When the leader receives a majority of *Response_Prepare* messages, it starts the *Accept* phase of consensus. However, if it receives at least one *Reject* message, it aborts the current round and it passes to the next one - lines 10-12 from Part 2 of Figure 18.

In the *Accept* phase, the leader selects the value to be proposed (lines 43-47, Part 1 of Figure 18). If the processes have already chosen one, it selects the same value, otherwise it selects its own value. Then, it sends an *Accept* message containing the proposed value to all the participating processes (line 49, Part 1 of Figure 18).

When a process receives the message it first checks if it is already committed to a higher round (line 40, Figure 17). If so, it sends a *Reject* message to the leader, informing it about the round number to which the process is committed. Otherwise, it stores the accepted value and sends an *Response_Accept* message to the leader (lines 43-45, Figure 17). When the leader receives a majority of responses, it enters the *Decide* phase and broadcasts a *Decide* message. With this message, it announces everybody that the current round is successful (line 7, Part 2 of Figure 18) .

However, the decided value for an instance could be lost on some processes, e.g. the leader crashes when broadcasting the decision, but the majority receives it. In this case, the process receives a *Prepare* message from an instance higher than its current one. When this happens, the process checks if it already has the decision for its current instance and determines which values that were decided between its instance and the current one are missing. If there is such a "gap", then it sends an *Ask_Decide* message to the current leader (lines 28-31, Figure 17). The leader answers with a *Stale* message, containing the missing decisions (lines 14-15, Part 2 of Figure 18).

The *Stale* message is also used to update a leader that is in an older instance than at least the majority of processes (lines 25-27, Figure 17). One example would be the case when a majority of processes have the decided value for an

[3] The set of processes that participate also includes the leader. This means that the leader sends the messages also to itself.

```
1: Variables:
2:    k    // Current consensus instance
3:    v_a ← null    // Accepted value
4:    r_a ← 0    // Round in which the value was accepted
5:    r_h ← 0    // Highest round number seen
6:    v_decided ← ∅    // Buffer of decided values
7:    future_rp ← ∅    // Stores the instance and the round number for
      future prepares.
8:
9:  propose(instance, proposal)
10:   k ← instance
11:   value ← proposal
12:
13: pass_to_instance(instance)
14:   k ← instance
15:   if future_rp.k = k and there is no pair (v,k) in v_decided then
16:      handle as if received message(Prepare,future_rp.k,future_rp.r)
17:      future_rp ← (0, 0)
18:   (v_a, r_a, r_h) ← (null, 0, 0)
19:
20: decide(k, proposal)
21:   wait until there is a pair (v, k) ∈ v_decided
22:   proposal ← v
23:
24: upon receive message(Prepare, k', r)
25:   if k' < k then
26:      send message(Stale, k, v_k, ...)
27:      return
28:   if k' > k then
29:      future_rp ← (k', r) where (k',r) are the highest seen so far
30:      if (v,k) or (v',k'), where k' is a next instance, are not in v_decided
         then
31:         send message(Ask_Decide, k)
32:      return
33:   if r < r_h then
34:      send message(Reject, k, r_h)
35:      return
36:   r_h ← r
37:   send message(Response_Prepare, k, r, v_a, r_a)
38:
39: upon receive message(Accept, k, r, v)
40:   if r < r_h then
41:      send message(Reject, k, r_h)
42:      return
43:   r_a ← r
44:   v_a ← v
45:   send message(Response_Accept, k, r)
46:
47: upon receive message(Decide, k, v)
48:   v_decided ← v_decided ∪ (k, v)
49:
50: upon receive message(Stale, k', v_k, ..., v_{k'-1}) with k' > k
51:   v_decided ← v_decided ∪ {(k, v_k), ..., (k' − 1, v_{k'-1})}
```

Figure 17: Pseudocode of the Paxos protocol on the process **p**.

```
1: Variables:
2:    k    // Current consensus instance
3:    phase    // Current phase of a consensus round
4:    v_l ← null    // Value to be proposed
5:    r_h ← 0    // Highest round number seen
6:    r_l ← 0    // The round number used by the leader
7:    n_rp ← 0    // Number of responses in the prepare phase
8:    n_ra ← 0    // Number of responses in the accept phase
9:    history ← ∅    // History of accepted values from previous rounds
10:
11: propose(instance, proposal)
12:    k ← instance
13:    value ← proposal
14:    wake-up leader
15:
16: pass_to_instance(instance)
17:    k ← instance
18:    if future_rp.k = k and there is no pair (v,k) in v_decided then
19:       handle as if received message(Prepare,future_rp.k,future_rp.r)
20:       future_rp ← (0, 0)
21:    (v_l, r_l) ← (null, 0)
22:    reset_round()
23:
24: reset_round()
25:    phase ← "Prepare"
26:    n_rp ← 0; n_ra ← 0
27:    history ← ∅
28:
29: leader()
30:    wait until woken-up
31:    if value not decided then
32:       reset_round()
33:       r_l ← r_h + 1
34:       send message(Prepare, k, r_l) to all p ∈ view
35:
36: upon receive message(Response_Prepare, k, r, v, r')
37:    if phase ≠ "Prepare" or r < r_l then
38:       return
39:    n_rp ← n_rp + 1
40:    if (v, r') ≠ (null, 0) then
41:       history ← history ∪ (v, r')
42:    if n_rp = (n + 1)/2 processes then
43:       select from history (v', r_v') with the highest r_v'
44:       if (v', r_v') = (null, 0) then
45:          v_l ← value
46:       else
47:          v_l ← v'    // There is no accepted value from previous rounds
48:       phase ← "Accept"
49:       send message(Accept, k, r, v_l) to all p ∈ view
50:
```

Figure 18: Pseudocode of the Paxos protocol on the leader - Part 1

```
1: upon receive message(Response_Accept, k, r)
2:    if r < r_l then
3:       return
4:    n_ra ← n_ra + 1
5:    if n_ra = (n + 1)/2 processes then
6:       phase ← "Decide"
7:       send message(Decide, k, v_l) to all p ∈ view
8:
9: upon receive message(Reject, k, r') for the first time
10:   if r' > r_l then
11:      r_h ← r'
12:      wake-up leader     // Start a new round
13:
14: upon receive message(Ask_Decide, k')
15:   send message(Stale, k, v_k', v_k'+1, ..., v_k−1)
```

Figure 18: Pseudocode of the Paxos protocol on the leader - Part 2

instance and have already passed to the next one, but the leader crashed when broadcasting the decision and the newly elected one has not received it.

Message Types The exchange of messages is depicted in Figure 19. The types of messages used during the protocol are the following:

- $Prepare(k, r)$ - Sent by the leader to all the processes from the view to announce the beginning of round r
- $Response\_Prepare(k, r, v, r')$ - Sent as a response to the prepare message from the leader. It contains the previous proposed value,v, and r' the round in which it was proposed. This is done to inform the leader that a value was accepted in a previous round, such that this value to be finally decided. It also shows the commitment of the process for the round r.
- $Accept(k, r, v)$ - Sent by the leader to all the processes. It contains the value that needs to be accepted, v, for the current round, r.
- $Response\_Accept(k, r)$ - It shows that the process has accepted the value for the round r.
- $Decide(k, v)$ - Sent by the leader to all the processes, to announce that the value v has been decided.
- $Reject(k, r')$ - Sent as a response to a propose or an accept from a round smaller than the current one. It has the purpose to inform the leader that the process is committed to a round r' that is higher than this leader's round.
- $Stale(k, v_k, ...)$ - Sent to a stale process or leader. It contains all the decisions taken between the stale instance and the current one.
- $Ask\_Decide(k')$ - Sent by a process to the leader when it receives a propose from a higher instance. Used to retrieve the lost decided values.

As a sequence of instances of consensus is used, every message contains the current instance number.

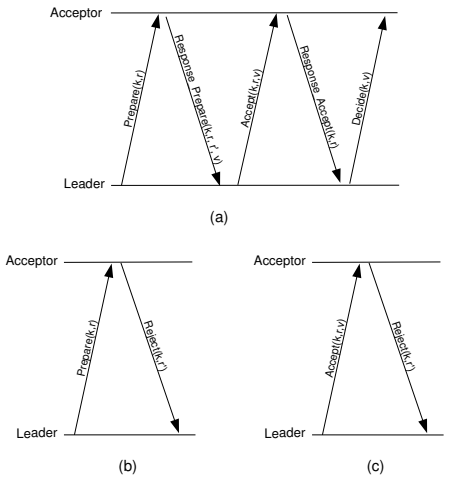


Figure 19: The types of messages exchanged during a round of Paxos. In (a) is presented the exchange of messages during a good run. (b) and (c) show the answer gave to a leader whose round r is smaller than the one known by the process (r')

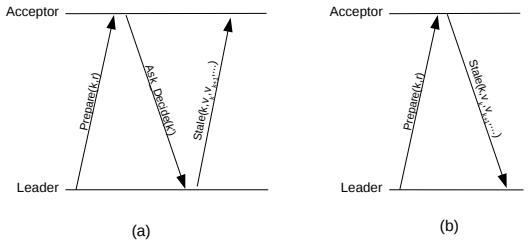


Figure 20: The types of messages exchanged during the Paxos protocol, to retrieve the decided values for lost instances.

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399